Abstract

This master thesis explores the possibilites and potentials of using a procedural approch based on L-systems to generate virtual cities. This report will show that our system, the ProCity Engine, can generate a pattern of streets and highways, divide land into allotments, and create appropraite buildings depending on the size and location of the corresponding lot. All this with only minimal input data in the form of a few image-maps and a couple of parameters.

For the creation of the pattern of streets and highways, as well as for the creation of buildings, L-systems have been extended to allow for external functions to modify the result after each iteration. The functionality of these L-systems can be separated from the ProCity Engine and used in its own right for other purposes.

The final stage of the master thesis consists of incorporating the ProCity Engine with an open source game engine, Crystal Space, to visualize the results of the generation process.

Index

1 Inledning

1.1 Introduktion

Detta dokument är en rapport, vilken omfattar ett examensarbete av två studenter på 20 poäng vardera, som utförts vid institutionen för datavetenskap på Lunds tekniska högskola. Examensarbetet genomfördes av Paul W. Florczykowski och Johan Hoberg, samt handleddes av Lennart Ohlsson, universitetslektor vid Lunds tekniska högskola.

Johan Hoberg tar examen i elektroteknik vid Lunds tekniska högskola, och har läst en kombination av elektronikkurser, programmering, hårdvarukonstruktion och datorgrafik. Han har ett djupare personligt intresse för allt kopplat till datorspel och hårdvara, men tycker även om de estetiska aspekterna av datorgrafiken.

Paul W Florczykowski tar examen i teknisk fysik från Lunds tekniska högskola och har ett brinnande intresse för programmering och då speciellt hans privata projekt inom artificiell intelligens. Han är också väldigt intresserad av algoritmteori och matematik i alla dess former. Givetvis finns det även här ett intresse för datorgrafik även ifall han inte anser de vara nödvändigt för människans fortlevnad.

Det som ingått i examensarbetet har varit c:a 5 veckors studier av åtskilliga dokument för att tillgodogöra sig den fakta som behövdes, sedan programmerande av en applikation i C++ under 14 veckor och slutligen skrivandet av en rapport under 5 veckor. Examensarbetet gjordes under perioden mars till september 2003.

Inspirationen till detta examensarbete, ett intresse för procedurell grafik, har sina rötter i ett gammalt amigaspel, Elite, där spelaren färdas genom en oändlig världsrymd, i vilken alla planeter skapas procedurellt när de korsar spelarens väg. Grafiken var givetvis simpel, men principen borde kunna tillämpas i dagens datorspel. Givetvis spelar den lavinartade utvecklingen inom datorgrafik också en viktig roll, eftersom ett examensarbete måste kännas meningsfullt för att något bättre resultat ska kunnas uppnås. Tanken på en ökande efterfrågan på arbetsmarknad, med 3D-grafik inom mobiltelefoni, på företags nätsidor, och mer och mer integrerat i människors vardag, gör att detta examensarbete känns betydelsefullt, och förhoppningsvis märks detta i nedanstående sidor, och i demoapplikationen.

1.2 Syfte

Modellering och visualisering av system gjorda av människor, så som stora städer, är en stor utmaning för datorgrafiken. Städer är system med hög funktionell och visuell komplexitet och de reflekterar historiska, kulturella, ekonomiska och sociala förändringar som skett genom åren. Genom att studera en stad som t.ex. New York kan stor mångfald märkas i gatumönster, byggnader, former och texturer. Tack vare utvecklingen inom datorer, med sådant som större minnen, snabbare processorer, och grafikprestanda som gått genom taket är det nu möjligt att modellera och visualisera stora städer med den mångfald som beskrivits ovan. Att modellera en stad är inte helt oproblematiskt, utan det finns mängder av variabler som måste tas i beaktning. Alla stadsmiljöer har ett vägnät som följer befolkningstäthet, miljöfaktorer och även speciella mönster som bestämts i förväg. Byggnaders utseende följer ofta olika historiska och estetiska mönster, men beror även på vilken funktion byggnaden har, samt i vilket område den är placerad. För att skapa en virtuell stad måste terräng, en vägkarta och mängder av byggnader genereras. För att generera en hel stad på detta sätt kan det lämpa sig att använda ett procedurellt angreppssätt.

Efter studier av tillgängliga procedurella städer visade det sig att de två som verkade ha kommit längst inom området är Yoav I H Parish och Pascal Müller med deras ”Procedural Modeling of Cities”[3]. Med denna vetskap valdes deras arbete som studieunderlag varefter försök skulle göras att implementera det som beskrivits om genereringen av procedurella städer. Deras dokumentation om arbetet är väldigt generellt och inte gjord för att någon direkt och enkelt ska kunna återskapa det som gjorts, men efter en längre tids studie av dokumentet har mycket av deras funktionalitet återskapats, samtidigt som mer funktioner och möjligheter har lagts till.

Det visade sig att stora delar av Parish och Müllers system baseras på användandet av L-system av olika slag, så som utökade L-system och stokastiska sådana. För att kunna implementera dessa funktioner krävdes studier av ett antal böcker så som ”The Algorithmic Beauty of Plants” [1] och ”Synthetic Topiary” [2].

Valet kring vilken spelmotor eller vilket renderingsverktyg som skulle användas föll på Crystal Space, dels därför att spelmotorn är gratis och tillgänglig för allmänheten, men även därför att institutionen för datavetenskap i Lund har stor erfarenhet inom området. Även här finns problem eftersom dokumentationen kring denna motor är minst sagt bristfällig, och mycket tid har lagts ned på att bemästra de funktioner som krävdes i arbetet.

Funktionaliteten som detta arbete innehar kan vara av största nytta inom flera områden. Inom t.ex. rollspel skulle det vara möjligt att skapa världar med ett oändligt antal städer vilket kan göra innehållet i spelen oändliga, om man dessutom applicerar det procedurella tillvägagångssättet på karaktärer. Inom first-person shooters och strategispel skulle det vara möjligt att varje match få en ny karta, något som skulle utveckla dessa båda typer av spel ytterliggare. Naturligtvis har slumpmässiga kartor genererats förr, men inte i tre dimensioner.

Även inom filmen kan det spara en väldig massa tid om alla städer inte behöver handmodelleras utan hela städer kan genereras med bara några parameterinställningar, men naturligtvis är applikationen också lämplig för 3D-grafiker och flera andra.

ProCity Engine är en utveckling av Parish och Müllers City Engine, med större funktionalitet och bättre möjligheter för andra examensarbeten att fortsätta förbättra och utveckla skapandet av procedurell geometri med hjälp av L-system.

1.3 Dokumentationsstruktur

Följande stycke tar upp hur dokumentationen är upplagd för att läsaren lättare skall kunna orientera sig och lokalisera det stycke som hon finner intressant. Teoriavsnittet i denna dokumentation tar upp all den information som krävs för att resten av detta arbete ska kunna tillgodogöras, varefter strukturen och funktionerna hos det som implementerats och programmerats under examensarbetet redovisas och förklaras. Slutligen diskuteras slutsatser och resultat, samt vilka delar av detta projekt som skulle kunna vidareutvecklas och förbättras.

Kapitel 2.1 tar upp grunderna i hur ett L-system fungerar, samt hur utökade och stokastiska L-system skiljer sig från det ursprungliga. Dessa används bland annat i genereringen av gator och byggnader och förståelse kring dessa är kritisk för att de matematiskt tunga delarna i detta arbete ska tillgodogöras.

Kapitel 2.2 tar upp CityEngine och ger ett förslag till hur Parish och Müllers dokument kan tolkas. Dels hur de föreslår att vägnätet ska byggas, men även hur byggnader och texturer ska genereras. Det är utifrån Parish och Müllers dokument som hela arbetet är grundat, och därför är det givetvis av största vikt att förståelsen för detta är total.

Kapitel 2.3 tar upp Crystal Space och diskuterar vilka funktioner denna spelmotor kan erbjuda, medan de rent tekniska detaljerna kring renderingen tas upp i kapitel 3. Kapitel 2.4 tar upp alternativ till Crystal Space och diskuterar varför dessa inte valdes.

Kapitel 3.1 till kapitel 3.6, samt 3.8 innehåller information kring själva ProCity Engine, där alla delarna i pipelinen tas upp individuellt och analyseras. Dessutom finns här information om alla mindre program som ingår, och tekniska detaljer kring Cystal Space skriptfiler.

Kapitel 3.7 tar upp de tekniska detaljer kring renderingen i Crystal Space som är kopplade till direkt C++ kod, samt vilka funktioner det visningsprogram som används har.

Kapitel 3.9 sammanfattar alla de skillnader som finns mellan Parish och Müllers CityEngine och ProCity Engine, vilka förbättringar som gjorts, samt vilka förenklingar som var nödvändiga.

Kapitel 4.1 visar de resultat som ProCity Engine producerar. Här finns olika screenshots samt diskussioner kring vad det är som egentligen gjorts. Kapitel 4.2 tar upp de detaljer i projektet som inte är fullständigt kompletta, samt lägger fram olika förslag till vidare utveckling av ProCity Engine. Slutligen följs detta av avslutande kommentarer och en sammanfattning i kapitel 4.3.

Appendix innehåller funktionslistor för mer teknisk information, samt vilka användardefinierbara variabler som finns tillgängliga.

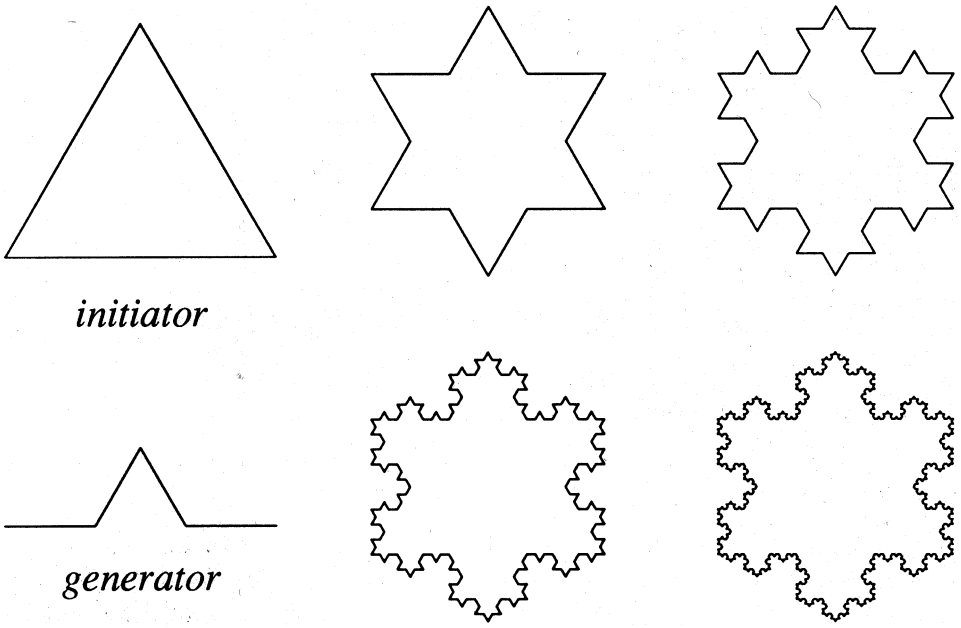
2 Teori

2.1 L-system

De tre sorters L-system som beskrivs nedan är av största vikt för att förstå hur ProCity Engine arbetar. Hela vägnätet bygger på det utökade L-systemet, och byggnadsgenereringen har sin grund i de stokastiska L-systemen. Det grundläggande L-systemet är givetvis av betydelse eftersom de två andra bygger på detta. Algoritmiskt sett är L-systemen den tunga biten i detta projekt, och ifall du bara har för intresse att använda applikationen, och inte förstå de underliggande principerna, så är detta kapitlet att hoppa över.

2.1.1 Grundläggande L-system

De förklaringar som förekommer nedan om vanliga L-system har till största delen hämtats från ”The Algorithmic Beauty of Plants” av Prusinkiewicz och Lindenmayer [1]. Det generella konceptet med L-system är omskrivning, d.v.s. genom att successivt ersätta delar av ett enkelt objekt med hjälp av en uppsättning regler produceras ett betydligt mer komplext object. Ett tidigt exempel som definierar detta är den så kallade snöflingekurvan som lades fram av von Koch året 1905. För en illustration av denna se figur 2.1.1, varefter en förklaring av proceduren följer.



Figur 2.1.1 – von Kochs snöflingekurva

Mandlebrot förklarar framtagandet av ovanstående figur med: ”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. Sådeles börjar varje steg av konstruktionen 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.”

Ovanstående var dock gjort på ett grafiskt objekt, och tiden visade att omskrivningssystem som arbetade på strängar kom att bli de mest framgångsrika och mest frekvent använda. Den förste som slog sig in på detta område var Thue, i början av föregående seklet, men det stora allmänna intresset för denna typ av system kom inte förrän i slutet 1950-talet, då Chomsky publicerade sitt arbete med en formell grammatik.

År 1968 introducerade en biolog vid namn Aristid Lindenmayer en ny typ av strängomskrivningsmekanism, som kom att kallas L-system, och grundades i hans intresse för

blommors struktur. Det var nämligen så att den största skillnaden mellan detta nya rön och Chomskys grammatik låg i att alla delar av en sträng bearbetades parallellt istället för sekventiellt. Paralleller kunde dras mellan detta system och hur utveckling sker i naturen, hos t.ex. blommor, d.v.s. alla celler utvecklas parallellt, inte i tur och ordning.

Den enklaste typen av L-system kallas DOL-system, och de är deterministiska och kontextfria. Dessa system är lämpliga för att exemplifiera grunderna på vilka det utökade systemet ska byggas, och enklast görs detta med ett upplysande exempel i tabell 2.1.1.

Initialsträng:	X
Regler:	y->xy
	x->y
Iterationssteg 1:	x->y
Iterationssteg 2:	y->xy
Iterationssteg 3:	xy->yxy
Iterationssteg 4:	Yxy->xyyxy

Tabell 2.1.1 – DOL-system

För att åskådliggöra ovanstående grafiskt går det att använda en virtuell sköldpadda, vilken man kan ge olika instruktioner. Dels går det att instruera sköldpaddan att flytta sig och rotera, men även att dra streck efter sig när den går. De fyra grundläggande instruktionerna följer i tabell 2.1.2 nedan, och gäller under förutsättningen:

”Givet en sträng v, utgångsläget 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.”

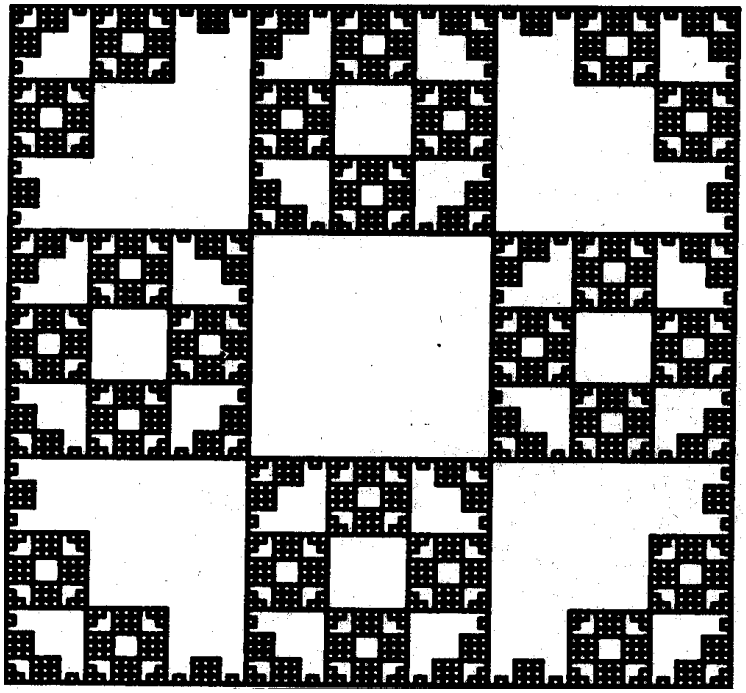
F	Förflytta dig framåt ett steg d. Sköldpaddan ändrar läge till (x’,y’, α), där x’=x+d*cosα och y’=y+d*sinα. Sköldpaddan ritar på så sätt ett linjesegment mellan punkterna (x,y) och (x’,y’).
f	Förflytta dig framåt ett steg d, men rita ingen linje.
+	Vrid dig vänster med vinkeln δ. Sköldpaddans nästa läge är (x,y, α+δ). Sköldpaddans positiva orientering av vinklarna är medurs.
-	Vrid dig höger med vinkeln δ. Sköldpaddans nästa läge är (x,y, α-δ).

Tabell 2.1.2 – Fyra grundregler för virtuell sköldpadda

Ovanstående regler skall nu demonstreras genom två enkla exempel, givit förutsättningar som beskrivs i tabell 2.1.3 och 2.1.4, vilket ger resultat som visas i figur 2.1.2 och 2.1.3.

n=4
δ=90°
Initialtillstånd: F-F-F-F
F->FF-F-F-F-FF

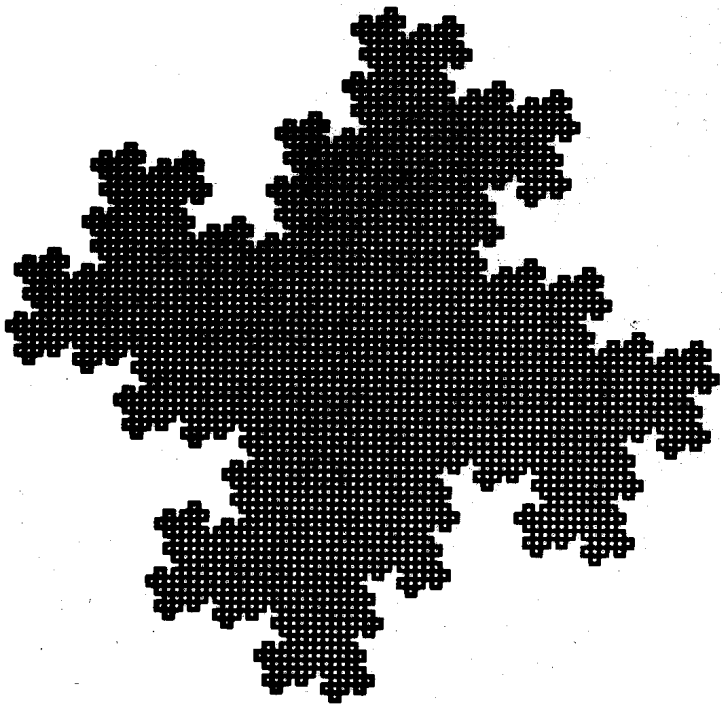
Tabell 2.1.3 – Grundförutsättningar för exempel 1



Figur 2.1.2 – Resultat av exempel 1

n=5
δ=90°
Initialtillstånd: F-F-F-F
F->F-FF--F-F

Tabell 2.1.4 – Grundförutsättningar för exempel 2



Figur 2.1.3 – Resultat av exempel 2

2.1.2 Utökade L-system

Det utökade L-system som föreslås följer konventioner som presenteras i "Synthetic Topiary" [2] av Prusinkiewicz, James och Mech. L-systemet består av ett antal kontextberoende produktionsregler. I grammatiken ingår även att varje produktionsregel får villkor samt en sannolikhet specificerad. Generella utseendet hos en produktionsregel ter sig:

id : lc <pred> rc : cond -> succ : prob

där:

id - regelns etikett
lc - kontexten på vänster sida
pred - den strikta föregångaren
rc - kontexten på höger sida
cond - villkor för tillämplighet
succ - efterföljaren
prob - sannolikhet för tillämpning (givet att p. regeln överhuvudtaget tillämpbar)

Den svenska nomenklaturen ovan är måhända något tveksam, men den åsido gäller att steget från L-system i sin enklaste form till denna utökade modell tilltalar förnuftet. En produktionsregel i det utökade L-systemet kan reduceras till:

pred -> succ

med samma konventioner som ovan, och då fås ett L-system i sin enklaste form. De ytterligare delarna i den utökade L-systemmodellen ger en förhöjd flexibilitet och uttryckskraft till produktionsreglerna. En produktionsregel som följer den ovan beskrivna syntaxen kan se ut så här:

p1: a < b(p) > c : p > 2 -> d(p-1) : 0.5

Regeln heter alltså "p1". Regeln är tillämpbar om det någonstans finns följande moduler efter varandra: " a b(x) c" och x > 2. Regeln säger i så fall att "b(x)" skall bytas ut mot "d(x-1)" med 50 % sannolikhet. T ex skulle "b(3)" bytas ut mot "d(2)".

Den återstående utbyggnaden av Lindenmayers ursprungliga metodik, bottnar i att parametrarna hos en modul (en modul är en enhetlig del av den datastruktur (sträng) som produktionsreglerna opererar på) kan lämnas odefinierade under själva omskrivningsfasen, dvs. ett iterationssteg. Efter det grundläggande iterationssteget då produktionsreglernas användning har lett till en ny sträng, utförs ett delsteg till då odefinierade parametrar tillskrivs värden. På så sätt kan L-systemet fås att ta hänsyn till omgivningen.

Det finns en syntax, enligt: Prusinkiewicz, James och Mech, som syftar till att ange en modul vars parametrar är odefinierade enligt själva produktionsfasen i ett iterationssteg. Det gäller att man får skriva ett frågetecken framför själva modulnamet. T ex skulle regeln:

a -> ?b(x),

i fall den tillämpas, producera en modul ?b(x), med odefinierad parameter.

Målsättningen i "Synthetic Topiary" [2] var att få växtliknande strukturer att anpassa sig till yttre former, som inte kan (på ett uppenbart sätt) beskrivas medelst produktionsregler. Parish & Muller har använt metodiken för att få utseendet hos ett gatunät samstämmigt med yttre villkor som: terränghöjd, befolkningstäthet, arkitektur.

2.1.3 Stokastiska L-system

Stokastiska L-system diskuteras i ”Synthetic Topiary” och de beter sig precis som vanliga eller utökade L-system, bortsett från att en regel inte alltid utförs, utan detta sker med en viss sannolikhet.

Först kontrolleras alltså om regeln kan tillämpas överhuvudtaget, d.v.s. om den möter alla de andra krav som ställs, varefter ett slumptal avgör om regeln tillämpas eller ej. Detta kan t.ex. användas för öka mångfalden bland växter som har gjorts med hjälp av L-system, och mycket annat inom procedurell grafik, så som att producera ett stort antal texturer med endast ett få utgångstexturer.

Nedan följer tre regler som alla har en viss sannolikhet att tillämpas. Bara en av dessa skall tillämpas, och ideologiskt sätt skulle checken göras på alla tre simultant men i praktiken göres kontrollerna i ordning, d.v.s. först kontrolleras om t.ex. regel 1 klarar sin sannolikhet och sedan kontrolleras regel 2, varpå kontroll av regel 3 följer. Om då t.ex. regel 2 har 30 % chans att tillämpas kommer dess sannolikhet att vara beroende på om regel ett tillämpades istället eller om regel 2 verkligen fick chansen att testa sin sannolikhet. Detta kommer att påverka de sannolikheter som är angivna så att dessa blir felaktiga, men detta kan motverkas genom att på ett slumpmässigt sätt välja vilken regel som ska kontrolleras först.

```
---------------------------------------
   ->Regel 1:  R1 -> R2 : 0.5
     Regel 2:  R1 -> R3 : 0.2
     Regel 3:  R1 -> R4 : 0.3
---------------------------------------
   ->Regel 2:  R1 -> R3 : 0.2
     Regel 3:  R1 -> R4 : 0.3
     Regel 1:  R1 -> R2 : 0.5
---------------------------------------
   ->Regel 3:  R1 -> R4 : 0.3
     Regel 1:  R1 -> R2 : 0.5
     Regel 2:  R1 -> R3 : 0.2

---------------------------------------
```

2.2 CityEngine

I detta kapitel beskrivs det tillvägagångssätt med vilket Parish och Müller procedurellt genererar en stad i ”Procedural Modeling of Cities” [3]. CityEngine som deras system kallas är kapabel att modellera en stad med ett begränsat antal inparametrar, och är till en hög grad kontrollerbart av användaren. Systemet skapar stadsmiljöer från ingenting, baserat på ett antal lättförståliga regler som kan utökas efter användarens behov. Det enda systemet behöver är en populationskarta och en karta som har information om sjöar berg och andra geografiska fenomen. Först analyseras hur vägnätet byggs upp från en funktionalistisk synvinkel, varefter en mer teknisk förklaring följer om hur vägnätet utvecklas med hjälp av utökade L-system. Efter detta beskrivs hur byggnader genereras med hjälp av stokastiska L-system samt hur platserna för byggnaderna beräknas. Slutligen diskuteras hur texturer genereras procedurellt utifrån speciella grundtexturer.

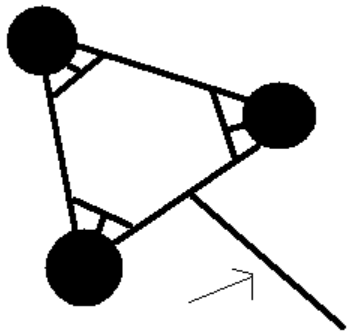
2.2.1 Vägnätsgenerering

Vägnätet som byggs upp av CityEngine går igenom nio produktionssteg, och efter att det gjort detta så har det utvecklats och blivit mer komplicerat. Detta upprepas sedan om och om igen, kalla de nio stegen för en produktionscykel, varpå vägnätet blir större och större, samt mer och mer invecklat. Med ett vanligt L-system hade denna utvecklig kunnat fortgå för evigt, men eftersom ett utökat L-system har lagt en mängd begränsningar för hur och var det får finnas vägar, kommer utvecklingen här efterhand att stagnera och till slut kommer de nio produktionsstegen inte alls att påverka vägnätet, varpå det utökade L-systemet inser att det är färdigt och processen är slutförd.

Nedan följer ett exempel på hur ett enskilt vägstycke utvecklas till ett vägnät medan produktionsstegen successivt utförs.

Startförutsättningar: Information om en vägbit och dess placering.

Produktionssteg 1: Om informationen om en vägbit på något sätt visar att vägbiten kommer att vara olämplig, vad gäller storlek eller placering, enligt globala begränsningar så raderas informationen om vägbiten i detta steg. Utan denna information kan produktionssteg två ej påbörjas och detta betyder att denna del av vägnätet har nått en gräns där det inte längre utvecklas. I figur 2.2.1 nedan representerar de svarta prickarna platser med hög befolkningstäthet, och de svarta strecken är vägar. Illustrationen visar på exempel när en väg kan tas bort p.g.a. att globala begränsningar säger att den är orelevant eller direkt olämplig. En pil pekar på den väg som kommer att tagas bort.

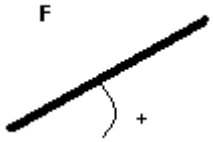


Figur 2.2.1 – Illustration av olämplig vägbit

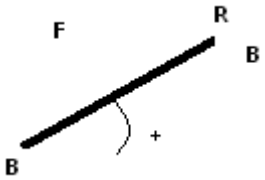
Produktionsteg 2: I detta produktionssteg sker själva skapandet av vägbiten, samt skapandet av information om vägbitar som senare ska utgå från denna vägbit. Först ges information om vägbiten; längd och vinkel från startposition. Sedan ges information om dels förgreningar i

båda ändar av vägbiten, samt information om fortsättning på vägbiten. Den information som fås om fortsättningen av vägbiten, är den information som kommer att användas i nästa produktionscykel.

Information om vägbit (F=vägens längd, + = vägens vinkel)

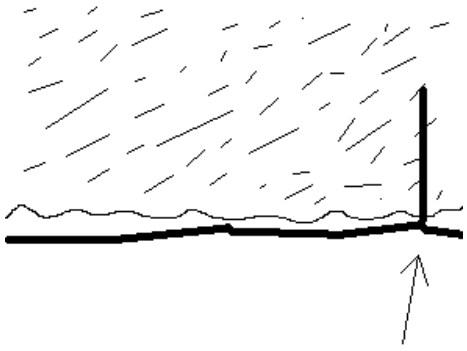


Sedan sätts information om var förgreningar kan ske (B) och information om hur vägen fortsätter, alltså information om nästa vägbit (R).



Det är i detta steg som de globala begränsningarna bestämmer om en väg är olämplig, och därför ska tagas bort i nästa produktionscykel av produktionssteg 1.

Produktionssteg 3: Här kontrolleras om vägbiten följer lokala begränsningar och om så inte är fallet tas vägbiten bort. Detta kan röra sig om att vägen korsar andra vägar på olämpligt sätt, går rakt ut i en sjö, eller kanske uppför ett berg. Nedan följer en illustration som visar en strandväg, och hur en förgrening av den kan vara olämplig. Det går givetvis att hitta anledningar till varför en väg skulle kunna gå rakt ut i vattnet, t.ex. att vägen då bildar en pir, men antag att så inte är fallet.



Figur 2.2.2 – Strandväg med otillåten förgrening

Produktionssteg 4: I detta produktionssteg räknas en räknare ner ett steg, och det är när denna räknare når noll som en förgrening sker. Detta produktionssteg gör inte själv förgreningen utan räknar bara ner så att nästa produktionssteg vet när det får göra något. Räknarens funktion är sådan att den ska fördröja förgreningar något så att inte de första vägstyckena dominerar hela kartan innan resten av vägnätet har hunnit etablera sig. Det hade kunnat arta sig så att p.g.a. att de första vägstyckena hunnit förgrena sig många gånger skulle de inte finnas någon plats kvar för påföljande vägstycken att förgrena sig p.g.a. att lokala begränsningar hade ansett att dessa kolliderade med redan existerande vägar.

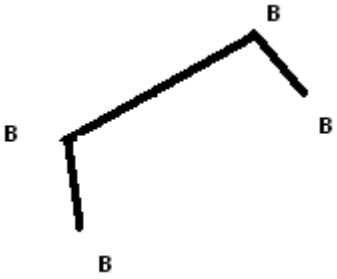
Produktionssteg 5: Eftersom produktionssteg fyra hela tiden räknar ner en räknare kommer denna räknare med jämna mellanrum att nå siffran noll och då ges information om hur en

förgrening ska ske i detta produktionssteg, som sedan kan realiseras en produktionscykel senare i produktionssteg 2. Ta vägstycke ovan och se vad som händer med det om det antas att information om förgreningar ges med en fördröjning på ett. Hänsyn tas här ej till att vägnätet hade fortsatt utan isolerar vägbiten och studerar förgreningen. B i skisserna visar var en förgrening ska ske, och R visar var ett nytt vägstycke kommer att skjuta ut.

Produktionscykel 1 - Produktionssteg 4: Fördröjning 1

Produktionscykel 2 - Produktionssteg 4: Fördröjning 0

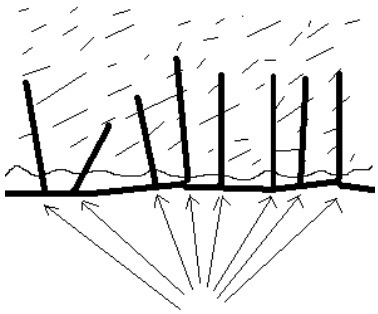
Produktionscykel 2 - Produktionssteg 5: Fördröjning 0



Produktionscykel 3 - Produktionssteg 2

I den sista skissen kan ses att de nya förgreningarna också skapar nya möjliga förgreningspunkter (B). I alla dessa skisser bortses från att varje nytt vägstycke, förutom att det har två förgreningspunkter (B), också direkt har information om påföljande vägstycke (R), som beskrivits i produktionssteg 2.

Produktionssteg 6: I produktionssteg 6 har de globala begränsningarna möjlighet att stoppa förgreningar genom att sätta den räknare som räknas ner i produktionssteg 4 till ett negativt värde. Om de globala begränsningarna t.ex. som i exemplet med strandvägen ovan säger att de inte vill ha några förgreningar alls, utan bara en enda lång väg, så har de här möjlighet att ta hand om det, så att de slipper hålla på och ta bort massa förgreningsvägar som hela tiden går ut i vattnet.



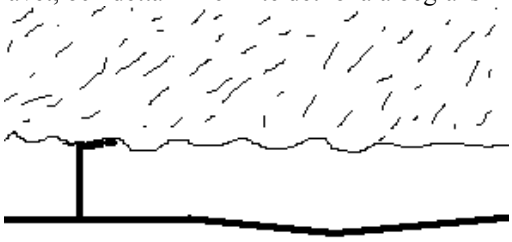
Figur 2.2.3 – Orimliga förgreningar

I figur 2.2.3 ovan visas vad som händer om man inte i produktionssteg 6 sätter räknaren till ett negativt värde. Detta gör att man i produktionssteg 3 hela tiden måste ta bort massa vägar som inte hade behövt skapas från början.

Produktionssteg 7: Om de globala begränsningarna i produktionssteg 2 bestämt sig för att en väg är olämplig, görs här vissa borttagningar för att det inte ska komma mer förfrågningar om att skapa denna väg. Detta steg används alltså mest för teknikaliteter.

Produktionssteg 8 och 9: Dessa har hand om själva utritandet av vägen, d.v.s. vägen som skapades i produktionssteg två, ritas upp om den har klarat sig igenom de krav som de lokala och globala begränsningarna ställde. Vissa vägar som inte riktigt passar, kan få sina attribut förändrade av de lokala begränsningarna så att de bättre smälter in i vägnätet. Se bilden nedan för exempel på detta.

Vägen går ut lite i havet, och detta finner inte det lokala begränsningarna accepterbart.



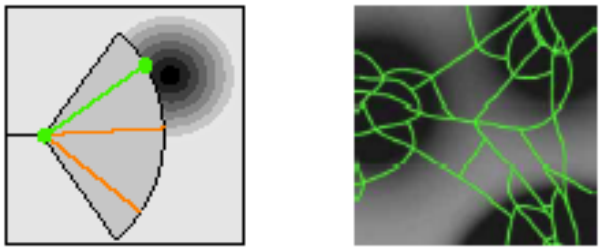
Lokala begränsningar böjer vägen efter kustlinjen och nu passar den mycket bättre.

Ett vanligt L-system skapar ett vägnät som bara bygger på en viss matematisk funktion, och då skapar ett nät av gator utan logik, medan ett utökat L-system också tar hänsyn till en massa andra variabler, lokala och globala begränsningar. Detta kommer att användas för att generera gatunätet i staden, och de lokala och globala begränsningarna kommer att kunna ställas in av användaren, för att få just det typ av gatunät som passar honom eller henne.

I ovanstående förklaring nämns globala och lokala begränsningar som sätts för att vägnätet ska få önskad form. De globala och lokala begränsningar som föreslås av Parish och Müller kommer nu att diskuteras och förklaras.

De globala begränsningarna kan delas in i två kategorier; populationsdensitet och

vägnätsmönster. Populationsdensitetsbegränsningen går ut på att motorvägar endast ska gå mellan områden med hög populationsdensitet, och inte fortsätta ut i ödemark. Detta görs genom att varje motorvägsslut skickar iväg strålar framför sig inom en inställbar halvcirkel, och sedan görs mätningar längs strålarna angående populationsdensitet. Alla mätningar viktas med avståndet från motorvägsslutet och sedan läggs de ihop, varpå motorvägen dras i den riktning där summan är högst. Figur 2.2.4 illustrerar ovanstående resonemang.



Figur 2.2.4 – Globala begränsningar angående populationstäthet

Ovanstående globala begränsning gällde för motorvägar, men det finns även regler som gäller båda sorters vägar, och dessa kan sammanfattas i olika vägmönster. Nedan följer en lista över de vägmönster som beskrivs av Parish och Müller i tabell 2.2.1.

Standard-regel	Detta är den enklaste möjliga regel, som inte följer något speciellt mönster utan bara följer populationsdensiteten.
New York-regel	Denna regel ställer speciella krav på vinklarna mellan gator, samt sätter en maxlängd och maxbredd på ett kvarter.
Paris-regel	Motorvägarna går i radiell led kring centrum av staden.
San Fransisco-regel	Gatorna följer här den väg som har minst lutning. Gator på olika höjdnivåer binds samman med mindre, kortare gator som går den väg som har högst höjdskillnad.

Tabell 2.2.1 – Globala vägnätsregler

Utöver globala begränsningar finns även lokala sådana, men dessa är inte mer komplicerade än vad som har förklarats ovan. D.v.s. följande steg görs för att kontrollera om en gata följer de lokala begränsningarna:

1. Kontrollera om vägsegmentet slutar på någon förbjuden position, så som i vatten, i parker, eller något liknande.
2. Kontrollera om gatan slutar för nära en annan gata. Om så är fallet ska de två gatorna kopplas samman.
3. Om gatan anses sluta på förbjuden position så försöker de lokala begränsningarna att göra om gatan så att den passar genom följande alternativ:
    - Minska längden på vägsegmentet med en viss faktor tills den passar inom det legala området
    - Rotera gatan så att den går längs med det förbjudna området. På detta sätt uppstår strandvägar, och vägar som går längs med parker.
    - Motorvägar kan tillåtas gå ut i vatten och in i parker till en viss gräns, och sedan ersättas med en bro eller tunnel vid rendering.

Nedan följer de regler som står att finna i Parish och Müllers ”Procedural Modeling of

Cities” [3] för de som är intresserade av de rent tekniska detaljerna av vägnätsgenereringen.

R - vägmodul
?I - frågemodul för insättning
B - förgreningsmodul
+ - sväng (med vinkel som argument)
F - rita framåt (med längd som argument)

Och dess moduler har följande argument, som sätts av *globalGoals* samt *localConstraints* - funktionerna:

R(del , ruleAttr)
?I(roadAttr , state)
B(del, ruleAttr, roadAttr)
+(angle)
F(length)

Där:

del - är en flagga som sätts i fall modulen skall raderas från systemet
ruleAttr - är en vektor med attribut som är regelspecifika
roadAttr - är en vektor med attribut som beskriver vägen
angle - är vinkel som "sköldpaddan" skall ställas i (absolut eller rel. ngt.)
length - är hur långt "sköldpaddan" skall röra sig

I det aktuella sammanhanget gäller att regler och attribut inte är specificerade in i minsta detaljnivå, principerna framgår dock tämligen klart ur Parish & Mullers text. Detaljutseendet måste naturligtvis specificeras då dessa algoritmer implementeras, t ex måste det då vara fastlagt vilka attribut *ruleAttr* omfattar osv.

Sammanlagt används nio produktionsregler och ett sk. axiom i Parish & Mullers system. Axiomet består av en initial vägmodul, som måste ligga i ett tillåtet område, samt en frågemodul för insättning. Låt oss betrakta axiomet med namnet omega:

```
omega: R(0, initialRuleAttr) ?I(inittRoadAttr, UNASSIGNED)
```

Regel 1, **Regel 2** och **Regel 3** styr R-modulen:

```
p1: R(del, ruleAttr) : del<0  ->  epsilon       (epsilon = den tomma strängen)

p2: R(del, ruleAttr) > ?I(roadAttr, state) : state == SUCCEED ->
                    +(roadAttr.angle) F(roadAttr.length)
                    B(pDel[1], pRuleAttr[1], pRoadAttr[1]),
                    B(pDel[2], pRuleAttr[2], pRoadAttr[2]),
                    R(pDel[0], pRuleAttr[0]) ?I(pRoadAttr[0], UNASSIGNED)

p3: R(del, ruleAttr) > ?I(roadAttr, state) : state == FAILED -> epsilon
```

pDel[0-2], pRuleAttr[0-2], pRoadAttr[0-2] sätts av *globalGoals* enl. de regler som beskriver globala målsättnigar för staden och/eller dess delar. **Regel 4** begränsar antalet förgreningar enl.:

```
p4: B(del, ruleAttr, roadAttr) : del > 0 -> B(del-1, ruleAttr, roadAttr)
```

Regel 5 skapar ett nytt vägsegment vid förgreningspunkterna, efter att fördröjningsräknaren når nollan:

```
p5: B(del, ruleAttr, roadAttr) : del == 0  ->  R(del, ruleAttr) ?I(roadAttr, UNASSIGNED)
```

Regel 6 tar bort förgreningspunkten i fall fördröjningsräknaren (nu anv. som en ”ta bort”-flagga) understiger nollan:

p6: B(del, ruleAttr, roadAttr) : del < 0 -> epsilon

Regel 7 tar bort frågemodulen, ifall motsvarande R-modul har fått en ”ta bort”-flagga:

p7: R(del, ruleAttr) < ?I(roadAttr, state) : del < 0 -> epsilon

Regel 8 och **Regel 9** ska avgöra, beroende på vilket värde parametern *state* har satts till, om vägen kan skapas eller om den ska tas bort (genom att ta bort frågeinsättningsmodulen)

p8: ?I(roadAttr, state) : state == UNASSIGNED -> ?I(roadAttr, state)

p9: ?I(roadAttr, state) : state != UNASSIGNED -> epsilon

2.2.2 Byggnadsgenerering

Efter skapandet av ett vägnät menar Parish och Mûller att de bebodda områdena skall delas in i något som de kallar ”blocks”, varefter dessa skall delas in i ”lots” vilka används för utplacering av byggnader. Det antas att de flesta av dessa områden är konvexa och rektangulärt formade, och systemet förbjuder därför skapandet av konkava sådana. Blocken delas med hjälp av en enkel rekursiv algoritm som delar de längsta sidorna som är relativt parallella, tills det indelade området är mindre än ett område angivit i förväg. Alla de områden som är för små eller inte har direkt angränsning till en gata tas bort från systemet, och det placeras inga byggnader där.

Det skall vara möjligt för användaren att avgränsa vissa områden och välja vilken höjd på byggnader som är tillåten inom detta område. Detta ger möjlighet att t.ex. begränsa skyskrapor till de centrala områdena av staden.

Alla byggnader i den virtuella staden modelleras med ett parametriskt stokastiskt L-system, och det genereras en byggnad i varje ”lot”. Byggnader delas in i tre kategorier: skyskrapor, kommersiella byggnader och bostadsområden. Dessa kategorier har olika produktionsregler, vilket gör att de får olika utseenden. Byggnader genereras genom att manipulera en grundläggande plan, och det görs med hjälp av ett stokastiskt L-system. Detta L-system består av ett antal moduler: transformationsmoduler (skala om och flytta), en modul som plockar ut oönskade delar, en förgreningsmodul och en avslutningsmodul, men även ett antal geometriska mönster för tak, antenner och liknande. Byggnadens slutliga utseende tas fram genom att grundstrukturen bearbetas med hjälp av de resultat som L-systemet producerar, och detta ger ett stort antal varianter av byggnader. Begränsningen i systemet är att byggnadernas funktionalitet inte kan uttryckas med endast ett fåtal enkla regler. Nedan följer en illustration av ovanstående resonemang.

Figur 2.2.5 - Fem steg i generering av byggnad

2.2.3 Texturgenerering

Ett vanligt sätt att lösa texturering av stadens byggnader är att använda detaljerade foton av riktiga fasader, scanna in dessa, modifiera dem, och sedan projicera dem på en lämplig sida av den geometriska byggnaden. Detta ger de mest detaljerade fasaderna, men det arbete som krävs för att lagra alla dessa texturer är allt för hög jämfört med tiden det tar att generera själva byggnaderna. Dessutom kan alltför många texturer orsaka problem med minnet. Dessa problem kan undvikas genom användning av procedurella texturer. Tyvärr kan dock inte all mindre detaljer på fasaderna modelleras på detta sätt. Vissa mönster så som tegel och trä kan visserligen analyseras och modelleras, men hela fasader, med fönster och dörrar, är inte lika enkla. Därför valdes att konstruera ett semiautomatiskt system för skapandet av fasader som utnyttjar ett flertal lager och en enkel sammansättningsteknik, som kallas nästlade rutmönster.

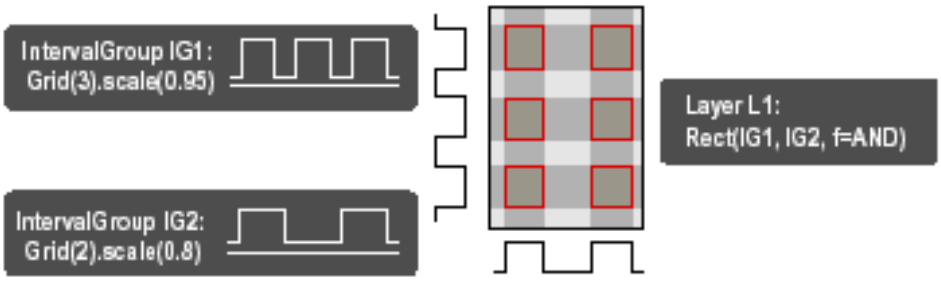
Systemet som ska beskrivas är baserat på ett antal antaganden kring fasader:

1. Fasader består av ett antal nästlade rutmönster, där nästan alla rutor har samma funktion, t.ex. dörrar eller fönster.
2. En enskild ruta påverkar storleken och positionen hos närliggande rutor. Som exempel kan nämnas att fönster på botten plan eller ovanför en dörr har olika storlekar.
3. Oregelbundenheter i rutmönstert påverkar hela rader eller kolumner och inte enskilda rutor

Målet med systemet var att skapa procedurella texturer som alla fångar en viss arkitektonisk stil vilket kan väljas av användaren genom att denne anger en allmän stiltextur. Utifrån varje sådan stiltextur ska det sedan gå att skapa en stor variation av texturer som alla har olika geometriska egenskaper, men ändå tydligt den angivna stilen. Det ska vara möjligt att använda inscannade bilder på bl.a. tegel, trä och fönster eftersom dessa ger en väldigt bra detaljrikedom.

Systemets grundläggande uppbyggnad är sådan att ett rutmönster är baserat på intervallgrupper, som består av ett antal icke överlappande, ordnade intervall. Fördelen med att använda endimensionella intervall som grunden för rutnätet är att rader och kolumner kan ändras bara genom att modifiera enskilda intervall i intervallgrupperna. Detta återkopplar också lämpligt till regel tre ovan.

Två intervallgrupper kan tillsammans bilda ett två-dimensionellt lager, som visas i figur 2.2.6 nedan. Ett lager består av två intervallgrupper, en *eval(s,t)* funktion och en *col* funktion. För varje *punkt(s,t)* där *eval(s,t)* returnerar en etta, anses denna punkt vara en *aktiv punkt*, och alla aktiva punkter i ett rutnät bildar det *aktiva området*. Om en punkt är aktiv returnerar funktionen *col* en färg eller någon annan egenskap för området. Icke rektangulära aktiva områden kan enkelt skapas genom att låta intervall i intervallgrupperna anta speciella funktioner.



Figur 2.2.6 – Tillverkning av procedurella texturer.

2.3 Crystal Space

En spelmotor kan sägas vara ett operativsystem för datorspel, och den är huvudkomponenten i den mjukvara som används för dessa. Ofta tar spelmotorn hand om sådant som kollisionsdetektering, hantering av skymda ytor, ljussättning, effekter, och AI. Gemensamt för nästan alla funktioner hos en spelmotor är att de på något sätt är kopplade till renderingen. Man skulle kunna säga att en spelmotor är allt det som inte spelproducenterna talar om, d.v.s. det som är gemensamt för spelen (till skillnad från t.ex. speciella texturer, bakgrunder, musik osv.). [d]

Crystal Space är ett stort opensource projekt, och det är ungefär 670 personer som har registrerat sig som utvecklare och användare varav många är aktiva, vilket märks på den mängd e-post som skickas fram och tillbaks mellan dem. Crystal Space är en gratis, portabel spelmotor som är skriven i C++ , och kan köras på de flesta plattformar, så som Unix, Linux, Machintosh eller Windows.[c]

De funktioner som stöds i Crystal Space följer i nedanstående lista, som delas in i sex olika kategorier. [a]

Generell arkitektur	Flexibelt plugin system som tillåter användning av andra moduler och skrip.
	Crystal Space använder sig av SCF för att kommunicera mellan de olika lagren, som till exempel mellan 3D-motorn och 3D-rastreraren.
	Stödjer 15/16 bitars true color och 32 bitars true color för bildskärmen.
	C++ källkoden är tillgänglig.
	Kan köra många olika upplösningar och är väldigt anpassningsbar via kommandotolken och via konfigurationsfiler.
Texturer och texturmappning	Texturer kan ha alla storlekar av tvåpotens och behöver inte vara fyrkantiga.
	Crystal Space stödjer mängder av filformat.
	Det finns många sätt att mappa en textur på en polygon, det finns möjligheter att skala om, rotera spegelvända och så vidare.
	Perspektivkorrigering med interpolation för varje 16 pixlar.
	Genomskinliga och halvgenomskinliga texturer för att skapa t.ex. fönster eller vattenytor.
	Mipmapping för att minska belastningen på texturcachen samt få texturer att se bättre ut på långt håll.
	Multitexturering med OpenGL.
	Dynamiska texturer.
Motor funktionalitet	Dynamisk gouroud skuggad himmelssfär och möjlighet till sol som rör sig över himlavalvet.
	Terrängmotor.
	Crystal Space stödjer speglar vilket

	gör det möjligt att skapa blänkande och reflekterande ytor.
	Statiska färgade ljuskällor med riktiga skuggor. Detta förberäknas innan världen visas.
	Dynamiska färgade ljus med mjuka skuggor.
	Förberäknad radiositet.
	2D sprites och ett partikelsystem som använder dessa.
	Dimma med olika färger som stöds både i mjukvara och hårdvara.
	Möjlighet att skapa atmosfäriska effekter kring solar och liknande.
	Stöd för böjda ytor med hjälp av bezier-kurvor och andra metoder.
	Ett synlighetssystem baserat på portaler, kd-träd och ”coverage” buffer.
	Hårdvaruaccelererade transformer för 3D triangel mesh sprites, böjda ytor och terrängmotorn.
	3D triangel mesh sprites med frame animation.
Portabilitet	Portad till Unix, GNU/Linux, Machintosh, Windows 32-bit
	MMX stöd för processorer som stöder detta.
	OpenGL hårdvaruaccelereringsstöd
Filformat	Crystal Space kan direkt ladda 3DS, MDL, MD2, ASE, OBJ och POV objekt.
	Banor kan lagras i zip-format.
	Det är möjligt att göra bibliotek av objekt, texturer och annat och lägga detta i en separat zip-fil.
	Möjlighet att konvertera MAP filer som används i Quake och HalfLife till Crystal Space.
Ytterliggare funktionalitet	Stöd för ljud samt 3D ljud.
	Stöd för flera olika ljudformat.
	En konsol finns tillgänglig precis som i Quake, och kan aktiveras med tab-tangenten.

Som nämnts ovan har Crystal Space stöd för att modellera och skapa en tredimensionell värld med hjälp av skriptfiler. En skriptfil kan beskrivas som en lista 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. Detta kommer att diskuteras ytterliggare och mer ingående i kapitel 3.7.

2.4 Renderingsalternativ

ProCity Engine är gjord för att procedurellt skapa en stad, som sedan ska visas med hjälp av Crystal Space, men med några smärre ändringar så kan staden renderas med något annat hjälpmedel, som t.ex. 3D Studio eller en annan spelmotor. ProCity Engine kan sägas vara uppdelad i två faser; en skapande fas och en visande fas. Den skapande fasen producerar ett recept som den visande fasen sedan renderar.

Om ett annat hjälpmedel än Crystal Space skulle vara att föredra är det därför möjligt att bara ersätta den visande fasen med en egen variant som då måste migreras med den skapande fasen. Det finns dock en begränsning, nämligen att det måste vara möjligt att skriva en skriptfil som sedan läses in av renderingshjälpmedlet och visas, d.v.s. strukturen på de filer som läses in av programmet måste vara tillgänglig för användaren.

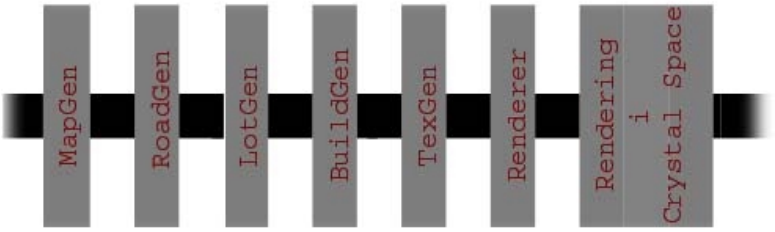
Tack vare denna struktur på ProCity Engine är livslängden för programmet längre än Crystal Space, och lätt att inkorporera i de lösningar som olika användare kan ha i åtanke.

3 Implementation

3.1 Övergripande struktur

ProCity Engine arbetar i ett antal väldefinierade steg när den procedurellt genererar en stad, och det är dessa steg som nedan kommer att beskrivas i detalj. Genom att användaren i en fil anger alla de variabler som finns tillgängliga kan hon styra hur staden kommer att genereras; storleken på staden, vägnätets utseende, topografin, husens utseende och andra egenskaper.

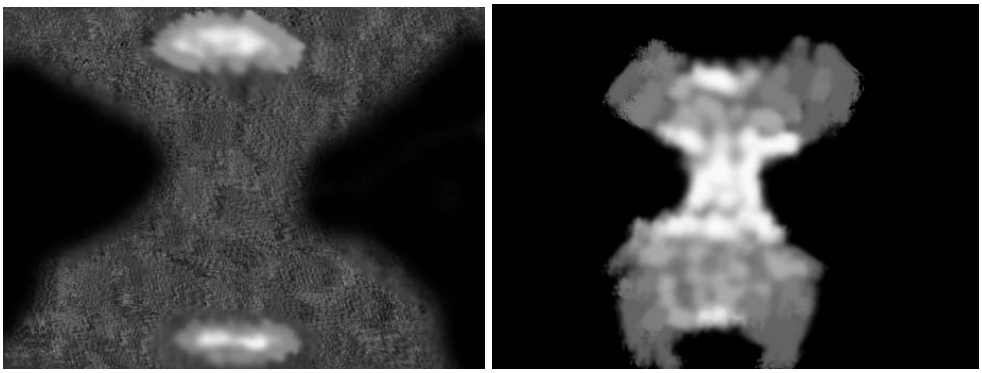
Först fastslås vilka höjd- och populationskartor som skall användas, varefter ett vägnät placeras ut beroende på dessa två kartor samt de regler som av användaren fastställts för att ge vägnätet en viss karaktär. När vägnätet är placerat delas hela kartan in i mindre områden, kallade lots, vilka var och en rymmer ett hus, en park eller något liknande. Sedan återstår bara att placera ut byggnader och andra objekt som ska finnas på kartan och slutligen ange vilka texturer dessa ska använda. Avslutningsvis skrivs all ovanstående information in i en Crystal Space skriptfil som sedan kan renderas med lämpligt verktyg. Nedan följer en illustration (figur 3.1.1) av den pipeline som utgör genereringsprocessen.



Figur 3.1.1 – ProCity Engine pipeline

3.2 MapGen

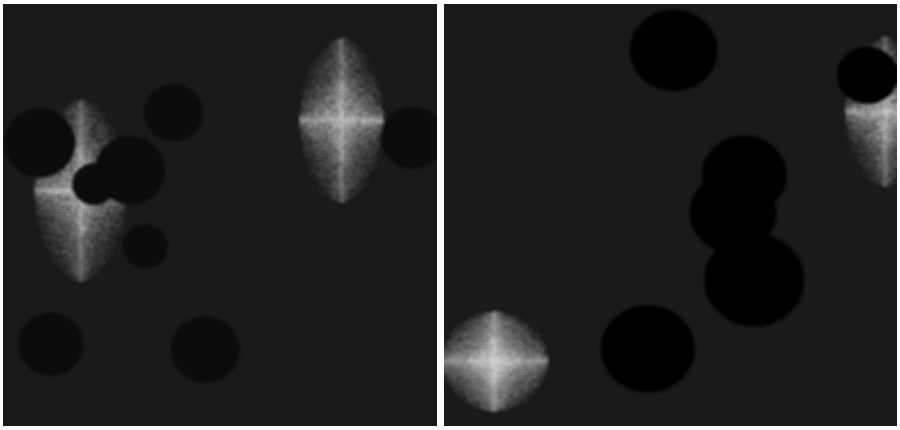
Det första steget i ProCity Engine pipelinen är att kontrollera om det finns några höjd- och populationskartor tillgängliga. Dessa ska vara av bitmap format men kan ha vilka färger som helst eftersom de kommer att översättas till sina gråskaliga motsvarigheter. Lämpligt är om kartorna håller normala proportioner, eftersom extrema kartor kan producera oönskade resultat. Exempel på lämpliga förgjorda kartor kan ses i figur 3.2.1.



Figur 3.2.1 – Förgjord höjd- och populationskarta

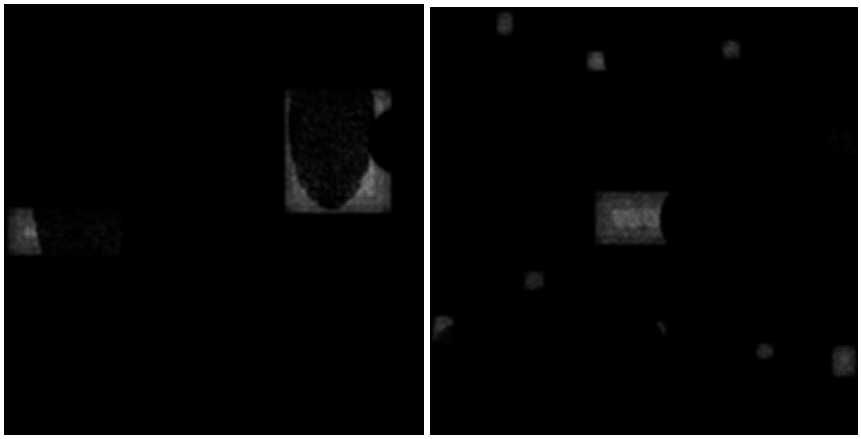
Om varken höjd- eller populationskarta finns angivna, så skapas dessa av ProCity Engine genom en slumpmässig process, men det går även att ange en av de två kartorna och få den resterande genererad.

Höjdkartan genereras slumpmässigt, baserat på användardefinierade parametrar som anger storleken i x och y led på kartan. Genereringen sker i tre enkla steg; först genereras marken, varpå denna sedan överlagras med vatten respektive berg. Exempel på genererade höjdkartor kan ses i figur 3.2.2.



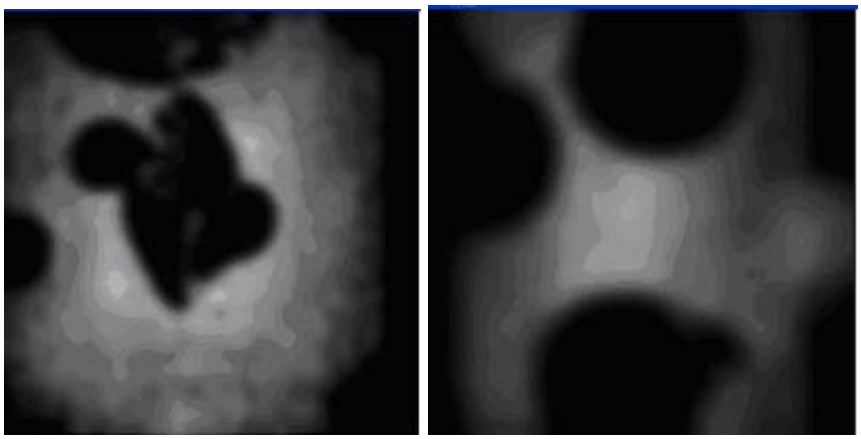
Figur 3.2.2 – Genererade höjdkartor

Populationskartan genereras i två steg, först placeras ett antal stora populationscenter ut, och sedan ett statistiskt sett större antal små populationscenter. När populationskartan genereras kommer denna att vara starkt korrelerad med höjdkartan genom att population inte tillåts i vatten, samt om lutningen är allt för hög, d.v.s. population tillåts inte heller uppe i berg. Populationskartor som baseras på de genererade höjdkartorna i figur 3.2.2, kan ses i figur 3.2.3.



Figur 3.2.3 – Genererade populationskartor

Det genererade resultatet var, särskilt i populationskartans fall, inte tillräckligt för annat än testningssyfte och krävde därför förbättring. Genom att zooma in på populationskartan och sedan låta den förstorade bildsektionen påverkas av höjdkartan producerades resultat som ansågs trovärdiga. Exempel på dessa nya populationskartor finns i figur 3.2.4 (observera dock att dessa inte har något samband med höjdkartorna ovan, utan endast demonstrerar det förbättrade resultatet) .



Figur 3.2.4 – Genererade populationskartor efter förstoring

Efter att kartorna genererats, bearbetas de var för sig av en funktion som jämnar ut snabba färgskiftningar och ger kartorna ett mjukare utseende. Detta måste ske för att berg inte ska se ut som stora stenpelare, utan ha naturliga sluttningar, och att populationsområden inte ska sluta alltför tvärt, utan successivt tunnas ut. Utjämningen baseras på två användardefinierade variabler för varje karta, en som anger hur många iterationer som ska ske, och en som anger hur många grannar i varje riktning en pixel ska ta hänsyn till när dess nya utjämnade värde beräknas. Beräkningen sker på följande sätt:

$$nypixel(x,y) = \left(\left(\sum_{a=-antalgrannar}^{a=+antalgrannar} \sum_{b=-antalgrannar}^{b=+antalgrannar} pixel(x+a, y+b) \right) \Big/ \left((\text{antalgrannar} * 2) + 1 \right)^2 \right)$$

För att bitmapfiler ska kunna skapas krävs ett API som möjliggör detta. Det finns med största säkerhet sådana tillgängligt, men för att underlätta skräddarsyddes ett enkelt API för att fylla precis de funktioner som behövdes i MapGen. APIet används senare även för RoadGen och LotGen, vilket framgår nedan.

3.3 RoadGen

Efter att en höjd- och populationskarta fastsällts ska ProCity Engine generera ett vägnät, och detta sker i RoadGen. I grunden ska RoadGen följa ungefär de specifikationer som angivits i Parish och Müllers ”Prodcedural Modeling of Cities”, men eftersom deras förklaringar är tolkningsbara finns det givetvis rum för en viss flexibilitet. Vissa aspekter av ProCity Engines RoadGen kan ses som en förlängning, eller förbättring, av det som föreslagits, och vissa funktioner och aspekter har helt enkelts skurits bort, för att de ansågs onödiga.

RoadGen bygger på ett parameterstyrt utökat L-system, och parametrarna kommer att beskrivas i kapitel 3.3.1, medan det utökade L-systemet beskrivs i kapitel 2.1.2. Att det är parameterstyrt gör att användare på ett enkelt sätt kan påverka processen i den riktning som önskas och det utökade L-systemet gör det enkelt att implementerar olika villkor på när vägar ska förgrenas och när de ska fortsätta.

Det L-system som används i RoadGen är dessutom självkänslig, d.v.s. det kommer inte att fortsätta iterera i evigheter och dra vägar ovan på varandra, utan det känner om två vägar är alltför nära varandra, och kopplar i så fall samman dessa och säger till systemet att den vägen inte längre ska fortsätta utvecklas, utan har nått sin slutliga form. Självkänsligheten kommer ytterliggare att diskuteras i kapitel 3.3.2 som samtidigt tar upp tidskomplexiteten för RoadGen och hur den kan variera beroende på självkänsligheten. Delar av L-systemets funktionalitet är dessutom portat till externa funktioner, precis som Parish och Müller gjorde, för att öka flexibiliteten hos systemet. Eftersom i Parish och Müllers fall lokala och globala begränsningar är externa funktioner gör det att det blir enklare att göra förändringar i dessa än om samma funktionalitet skulle implementeras med ett antal komplicerade regler, som sedan hade varit betydligt jobbigare att ändra på. Dessa externa funktioner i RoadGen kallas branchgoals och contgoals och beskrivs ytterliggare i kapitel 3.3.3.

Förutom att RoadGen är beroende av ett antal användardefinerade parametrar är den även beroende av en höjd- och en populationskarta. Dessa två har antingen genererats av MapGen eller gjorts av användaren och de används av de externa funktionerna för att avgöra var det är tillåtet att förgrena och fortsätta vägar. Hur detta fungerar beskrivs ytterliggare i kapitel 3.3.3.

RoadGens funktion är förutom att generera ett vägnät också att på något sätt förmedla detta vägnät till nästa instans i pipelinen och detta gör den genom att skapa en textur som är en bild av landskapet med vägnätet överlagrat. Denna textur används sedan av renderer och i skriptfilen mappas den på terrängen som skapas med hjälp av Crystal Space terräng motor. Förutom denna textur skapas också en del intressanta kartor som inte skickas vidare till nästa pipeline steg utan visar information om det vägnät som bildats. Motorvägsdensiteten och gatudensiteten kartläggs och presenteras i bilder där vägarna har överlagrats med ett rutmönster, där en vit ruta betyder hög vägdensitet och en svart sådan betyder inga vägar alls. När vägnätet successivt byggs upp, uppdateras dessa kartor hela tiden och om en ny väg ska dras i ett område kontrolleras så att detta område inte har alltför hög vägdensitet redan för i så fall tillåts inte vägen. Motorvägsdensitet och gatudensitet är båda användardefinierbara parametrar som kommer att beskrivas i kapitel 3.3.1.

3.3.1 Parameterförklaring

Nedan följer en förklaring av de användardefinerade parametrar som finns tillgängliga i RoadGen. De som står i förklaringen är speciellt för motorvägar, men parametrarna för vanliga gator är analoga. Efter förklaringen av parametrarna följer två exempel på vilka gatunät olika parametrar kan bilda, detta för att ge exempel på effekterna av att tilldela parametrar vissa specifika värden. Varje parameter förklaras nedan individuellt och med kortare information om hur vägnätet förändras beroende på vilket värde dessa har.

int const hw_type = HIGHWAY;
Denna variabel är en flagga som anger vilken begränsningsmodell som skall användas. Som det är just nu ska denna alltid vara satt till HIGHWAY för motorvägar, men det är möjligt för framtida utveckling att göra ytterligare begränsningsmodeller om så önskas och i så fall skriva in någonting annat här.

float const hw_rule = 0;
Denna variable anger vilken regel som skall användas för att bestämma i vilken riktning som motorvägen skall fortsätta. Den regel som finns implementerad, regel 0, är sådan att motorvägen följer den riktning som har högst populationsgradient, men framtida utveckling kan skapa andra regelverk för denna process. T.ex. kan motorvägar följa den minsta höjdgradienten.

float const hw_seclen = 30; [Litet värde = 10 |Normalt värde = 35 |Stort värde = 100]
Denna variabel anger hur lång varje nytt vägstycke som läggs till vägnätet är. Om dessa är långa kan vägnätet lätt få en ojämn karaktär, eftersom små steg ger en betydligt mer jämn övergång när vägen byter riktning p.g.a. reglerna. Om denna variabel väljs alltför liten, för att få väldigt mjuka övergångar, så kommer genereringstiden att ökar därefter.

float const hw_rsqrsize = 100; [Litet värde = 30 |Normalt värde = 50 |Stort värde = 200]
Detta är en variabel som avgör hur grov vägdensitetskartan är. Motorvägar ska t.ex. inte tillåtas gå alltför nära andra motorvägar och därför räknas vägdensiteten i stora områden, d.v.s. då är denna variabel stor. Gator däremot ska få ligga betydligt tätare och då räknas vägdensiteten i betydligt mindre områden, d.v.s. då är denna variabel liten.

float const hw_plan = 500; [Litet värde = 100 |Normalt värde = 600 |Stort värde = 1500]
Denna variabel anger hur lång framför sig en väg ska kontrollera, i regel 0's fall, populationsdensiteten för att avgöra i vilken riktning nästa vägstycke ska dras. Med ett väldigt stort värde här kommer vägarna generellt att bättre följa de stora populationsområdena, men det finns dock fall när ett mindre värde kan ge bättre resultat. T.ex. kan det finns två distinkta populationsområden som inte ska vara sammankopplade, kanske skiljs de av vatten eller allt för höga berg, och då är ett mindre värde bättre för att varje enskilt populationsområde inte ska påverkas av andra populationsområde som inte har någon koppling.

float const hw_turn = 6; [Litet värde = 0 |Normalt värde = 8 |Stort värde = 30]
Denna variabel anger det maximala värdet i grader som en vägsektion får svänga jämfört med tidigare vägsektion.

int const hw_bchdangle = 10; [Litet värde = 0 |Normalt värde = 20 |Stort värde = 50]
int const hw_bchdangle_neg = 10; [Litet värde = 0 |Normalt värde = 20 |Stort värde = 50]
Dessa variabler anger det maximalt positiva respektive negative värde som förgreningar från en väg får avvika från 90 grader. D.v.s. om dessa värden båda är 10, tillåts förgreningar från en väg att sticka ut i en vinkel mellan 80 till 100 grader. Är båda dessa värden noll så går alla förgreningar rakt ut från modervägen.

float const hw_bchdist = 150; [Litet värde = 100 |Normalt värde = 300 |Stort värde = 500]
Denna parameter anger med hur stort avstånd förgreningar får ske. Om värdet är stort sker väldigt få förgreningar, och om värdet är litet fås ett väldigt tätt mönster av gator. Naturligtvis är det så att ett litet värde här ökar genereringstiden avsevärt.

float const hw_enddist = 29; [Normalt värde = bchdist - 1 |Stort värde = 2*bchdist]
Denna variabel anger hur långt från slutet av ett nyligen placerat vägsegment som en förgreningspunkt får vara innan dessa två kopplas samman och både förgreningspunkten och punkten där vägen skulle fortsätta markeras

som förbrukade.

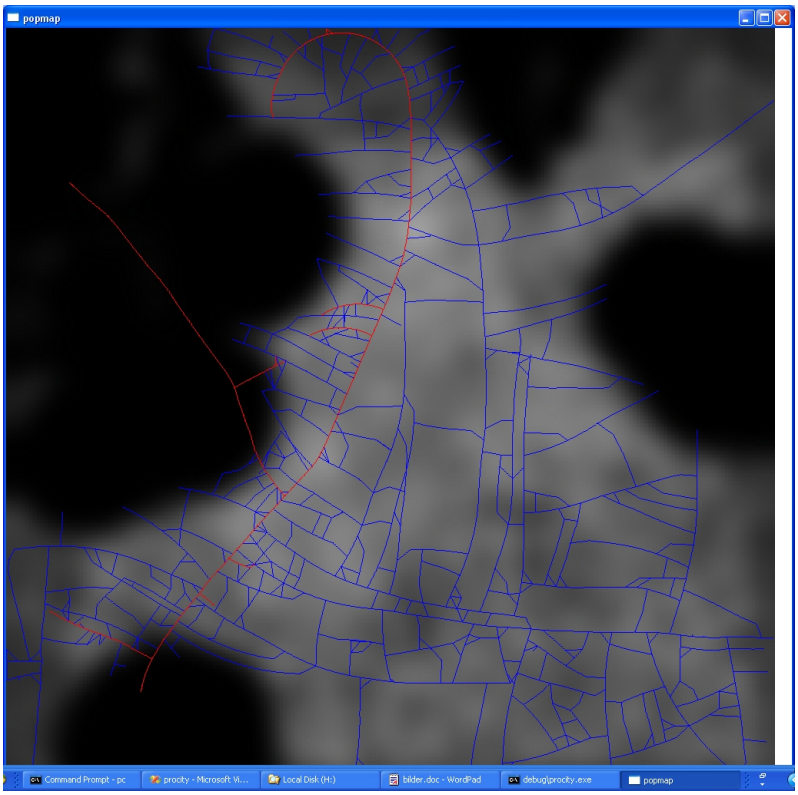
int const hw_bdel_low = 1; [Litet värde = 1 |Normalt värde = 5 |Stort värde = 20]
int const hw_bdel_high = 50; [Litet värde = 3 |Normalt värde = 10 |Stort värde = 100]
Dessa variabler är övre och nedre begränsningar på hur länge en förgreningspunkt kan bli fördröjd innan den tillåts utvecklas till en väg. Med höga värden här tillåts att vägar fortsätter längre innan de blockeras och tvingas kopplas samman med förgreningar från andra vägsegment. Om dessa sätts alltför lågt sprids en massa förgreningar väldigt tidigt och blockerar för huvudvägen så att gatunätet blir tätt och centrerat till vissa områden.

float const hw_existenz = 0.0001; [Litet värde = 0,0001 |Normalt värde = 0,01 |Stort värde = 1]
För gator antyder denna variabel hur låg populationsdensistet som är lägsta möjliga tröskel för att ett vägsegment ska tillåtas i det området. Detta hindrar en massa gator från att sprida sig i ödemarken om man sätter denna variabel rätt. Motorvägar tillåts inte längre fortsätta om kvoten mellan den aktuella popultionsgradienten, som motorvägen följer, och den nya populationsgradienten, som motorvägen är tänkt att följa, är alltför låg.

float const hw_creation_ratio = 0.05; [Litet värde = 0,001 |Normalt värde = 0,1 |Stort värde = 0,9]
Denna variabel är gränsen för hur hög vägdensiteten får vara inom ett visst område för att det fortfarande ska tillåtas att nya vägar skapas där.

float const hw_max_elev = 10; [Litet värde = 0,01 |Normalt värde = 0,1 |Stort värde = 1]
Denna variabel anger den största möjliga lutning som en väg får ha innan den tas bort. Detta är för att undvika att vägar går rakt upp och över berg.

Nedan följer två exempel på hur kartor kan se ut, samt vilka parametrar som användes för att generera dessa.

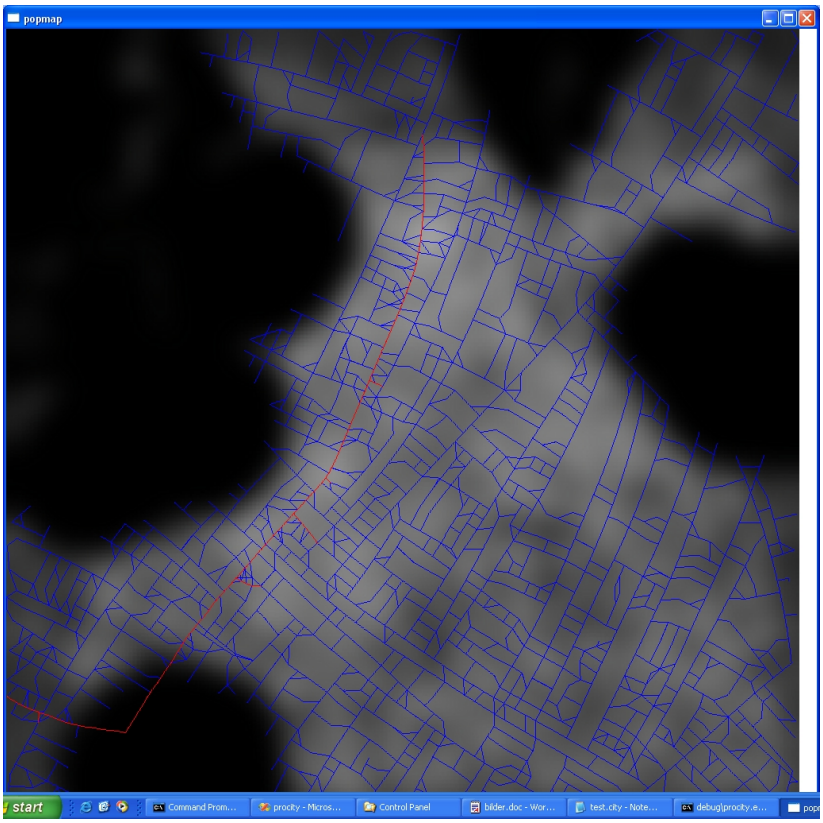


//Motorvägar
hw_type 0
hw_rule 0
hw_seclen 15
hw_rsqrsize 100
hw_plan 500
hw_turn 5

hw_bchdangle 10
hw_bchdangle_neg 10
hw_bchdist 400
hw_enddist 13
hw_bdel_low 10
hw_bdel_high 30
hw_existenz 0
hw_creation_ratio 0.1
hw_max_elev 100

//Gator
st_type 1
st_rule 0
st_seclen 30
st_rsqrsize 30
st_plan 50
st_turn 3
st_bchdangle 15
st_bchdangle_neg 15
st_bchdist 50
st_enddist 29
st_bdel_low 5
st_bdel_high 15
st_existenz 0.00005
st_creation_ratio 0
st_max_elev 100

Till skillnad från det första exemplet har följande betydligt hårdare parameterinställningar, som näst intill tvingar vägarna att bilda ett rutnät.



//Motorvägar
hw_type 0

hw_rule 0
hw_seclen 15
hw_rsqrsize 100
hw_plan 500
hw_turn 5
hw_bchdangle 10
hw_bchdangle_neg 10
hw_bchdist 400
hw_enddist 13
hw_bdel_low 10
hw_bdel_high 30
hw_existenz 0
hw_creation_ratio 0.1
hw_max_elev 100

//Gator
st_type 1
st_rule 0
st_seclen 30
st_rsqrsize 30
st_plan 50
st_turn 1
st_bchdangle 3
st_bchdangle_neg 3
st_bchdist 30
st_enddist 29
st_bdel_low 3
st_bdel_high 5
st_existenz 0.00005
st_creation_ratio 0
st_max_elev 100

3.3.2 Komplexitetsanalys

I nedanstående kapitel beskrivs tidskomplexiteten för RoadGen och hur denna påverkas om RoadGen är självkänsligt eller inte. Självkänsligheten i systemet uttrycker sig på sådant sätt att varje nytt vägsegment känner av resten av vägnätet och anpassar sig därefter. Nedan beskrivs fyra aspekter som ingår i självkänsligheten hos systemet.

Om ett vägsegment märker att det korsar ett annat vägsegment, så anpassar det sig på så sätt att det ansluter sig till det redan utlagda vägsegmentet och på så sätt bildar en sluten loop, vilket avslutar just denna vägs utveckling. Detta motverkar uppkomsten av onaturliga korsningar, samt hjälper till att få systemet att konvergera.

Ytterliggare en funktionalitet som inte skiljer sig avsevärt, rent praktiskt sett, från vägkorsningskontrollen är förgreningspunktskontroll. Genom att kontrollera om slutet på vägsegmentent är i närheten av en förgreningspunkt på någon annan väg, och då koppla samman dessa, uppkommer ytterliggare ett sätt att avsluta vägar, vilket även detta minskar antalet onaturliga korsningar, samt får systemet att konvergera.

Vidare finns två funktioner som kontrollerar mängden vägar och motorvägar inom ett visst område och bara tillåter att nya vägar skapas här, om den nuvarande mängden inte överstiger ett visst tröskelvärde. Dessa två funktioner gör att det inte blir en onaturligt hög koncentration av gator på vissa ställen, utan att antalet gator överensstämmer med den population som finns i området.

Med definitionen av självkänslighet klarlagd ska det nu diskuteras hur detta påverkar tidskomplexiteten för programmet. För att kunna klarlägga tidskomplexiteten kommer följande termer att användas:

N - Antal moduler
A - Antalet aktiva moduler
K - Tidskomplexiteten

Antalet moduler i ett L-system är storleken på systemet och antalet aktiva moduler är de moduler i L-systemet som fortfarande kan skrivas om och anta nya former. Det förutsätts givetvis att termen tidskomplexitet är känd.

I fallet utan självkänslighet gäller följande resonemang för att bestämma tidskomplexiteten:

- Varje modul tar en konstant tid att bearbeta eftersom en fix funktion anropas.
- Om minneshantering bortses från, gör detta att tiden det tar är proportionell mot antalet moduler.
- Detta medför att tidskomplexiteten då blir K = O(N).
- Tiden som går åt i minneshantering är svår att avgöra eftersom windows minneshantering inte är lättöverskådlig, men det kan antas att K=O(N^x), där x är en exponent av okänd storlek, möjligen 2.

I fallet med självkänslighet gäller följande resonemang för att bestämma tidskomplexiteten:

- För antalet inaktiva moduler gäller samma resonemang som ovan.
- Alla aktiva moduler måste kontrolleras med alla andra moduler och detta ökar tidskomplexiteten.
- Detta medför följande tidskomplexitet, K = O((N-A) + A*N), utan minneshantering.
- Med minneshantering blir detta i storleksordningen K = O(A*N + N^x), där x är en exponent av okänd storlek.

För att minska den tid det tog för systemet att köra färdigt gjordes en minneshanteringsoptimering. I stället för att vid varje iteration gå igenom varje modul, bearbetades bara de moduler som fortfarande var aktiva, och detta gav uppskattningsvis följande tidskomplexitet:

- Tidskomplexiteten K = O(N*A + A^x), där x är en exponent av okänd storlek.
- Detta visade sig rent praktiskt vara betydligt snabbare p.g.a. att A << N.

Det antagande som görs ovan är att minneshanteringen i Windows kan antas ha en polynomisk tidskomplexitet, eftersom det finns generella algoritmer som gör minnes hanteringen polynomisk även för mer avancerade system. Dock är kunskapen begränsad inom Windows minneshantering och det är möjligt, men mindre troligt, att minneshanteringen skulle vara mer krävande.

Ovanstående utredning avser tidskraven för ett iterationssteg. Antalet iterationssteg som krävs för att generera en hel stad beror av ett antal faktorer, som till exempel: hur populationskartan är beskaffad, hur stor staden är, vilka de övriga parametrarna är osv... Man skulle kunna försöka formulera ett mera allmänt uttalande, t ex: ”antalet iterationssteg växer med befolkningen”. Sådana uttalanden, även om i något mån sanna, är alltför generella för att kunna tillföra komplexitetsanalysen något värdefullt.

3.3.3 Externa funktioner, moduler och produktionsregler

Detta kapitel inleds med förklaringar kring de moduler som ingår i RoadGen, varefter de produktionsregler som används för att generera vägnätet beskrivs. Slutligen diskuteras de externa funktioner som ingår i programmet, hur dessa skiljer sig från de som Parish och Müller föreslog och varför ProCity Engines system är mer uttryckskraftigt och flexibelt.

Inledningsvis skall de moduler som används i RoadGen framställas och förklaras. Det finns fyra olika sorters moduler och dessa är som följer:

- R(state, rdattr)
- B(del, rm, rdattr)
- RD(rdattr)
- BD(rm, rdattr)

Den första modulen beskriver ett vägstycke som fortfarande är i utveckling, och den tredje beskriver ett vägstycke som har nått sin slutliga form, d.v.s. från den kommer inte ytterliggare vägar att förlängas eller förgrenas. Den andra modulen i listan ovan beskriver en förgrening och den fjärde beskriver en förgreningspunkt som det inte kommer att uppstå nya vägsegment ifrån. De attribut som ovanstående moduler innehar kommer att beskrivas i en lista nedan:

- "State" kan ha tre värden och attributet beskriver om vägstycket ska tas bort ("fail"), om den har slutat utvecklas ("final"), eller om den kommer att fortsätta utvecklas ("ok").
- "rdattr" innehåller information om de två punkter som vägsegmentet går mellan.
- "del" kan ha flera olika värden, som ger upphov till olika effekter. Om värdet är större än noll betyder detta att en fördröjning med det aktuella värdet ska ske. Om värdet är noll betyder det att förgrening ska ske, och om den är mindre än noll ska B modulen ersättas med en BD modul.
- "rm" är ett attribut som behövs för att avgöra ifrån vilken av de två punkterna på ett vägstycke som en viss förgrening ska ske.

Kunskap om ovanstående beskrivna moduler behövs när produktionsreglerna nu ska analyseras. De kommer att beskrivas i ordning med en förklaring följande varje regel.

Regel 1: R(state, @rdattr) : state == ok -> RD(rdattr) B(@del1, 1 , rdattr) B(@del2, 2, rdattr) R(@new_state, @new_rdattr)

- Regel 1 säger att om ett vägstyckes "state" har värdet "ok" så ska vägstycket ersättas med ett vägstycke som är färdigutvecklat, två förgreningspunkter samt ett nytt vägstycke. När det står ett @ tecken före ett attribut betyder detta att en extern funktion tar in värdet (om värdet står till vänster om likhetstecknet) eller anger värdet (om värdet står till höger om likhetstecknet). I regel 1 är det den externa funktionen "contgoals" som gör detta, och denna funktion måste anges omedelbart efter att regeln angivits och innan nästa regel anges, för att programmet inte ska haverera.

Regel 2: R(state, rdattr) : state == fail -> BD(rm, rdattr)

- Regel 2 säger att en modul R som vars attribut "state" har värdet "fail" skall ersättas med en modul BD. Med andra ord ska ett vägstycke ersättas med en förgreningspunkt

som har förgrenats färdigt, d.v.s. något nytt vägstycke skall inte läggas här av någon anledning.

Regel 3: R(state, rdattr) : state == final -> RD(rdattr) BD(rm, rdattr) BD(rm, rdattr)

- Om attributet istället har värdet "final" ska vägstycket ersättas med ett färdigutvecklat vägstycke samt två färdigutvecklade förgreningspunkter. Detta betyder att det aktuella vägstycket skall placeras, men att det ej kommer att bildas några ytterliggare vägar från denna, utan att detta är en återvändsgränd eller att vägen kopplats samman med någon annan väg.

Regel 4: B(del, @rm, @rdattr) : del == 0 -> R(@new_state, @new_rdaatr)

- Denna regel säger att en förgreningspunkt med attributet "del" vars värde är 0 skall förgrenas och låta ett nytt vägstycke skjuta ut från punkten.

Regel 5: B(del, rm, rdattr) : del > 0 -> B(del-1, rm, rdattr)

- Denna regel säger att om en förgreningspunkt vars attributet "del" har ett värde som är större än noll så ska denna inte låta något nytt vägstycke skjuta ut från punkten utan fördröja detta tills vidare.

Regel 6: B(del, rm, rdattr) : del < 0 -> BD(rm, rdattr)

- Regel 6 säger att en förgreningspunkt med attributet "del" vars värde är mindre än noll skall ersättas med en förgreningspunkt som inte längre har möjlighet att förgrena sig utan vars utveckling är klar.

Slutligen kommer de externa funktionerna att studeras och beskrivas i detalj. I stället för att ha en uppdelning som Parish och Müller i "Global goals" och "Local constraints" så har ProCity Engine en uppdelning i "Branch goals", "Cont goals" och "Local constraints". Nedan kommer var och en av dessa att förklaras i detalj, men grovdraget tar "Cont goals" hand om i vilken riktning vägar fortsätter, medan "Branch goals" tar hand om hur och var förgreningar uppstår. "Local constraints" används sedan för att ta bort de vägar som hamnat på icke tillåtna positioner.

"Cont goals" är den externa funktion som avgör i vilken riktning en väg ska fortsätta efter det att ett vägsegment har placerats ut. Här finns givetvis möjlighet att ange precis vilka regler som skall gälla, men de som finns implementerade i ProCity Engine i nuläget är att en väg försöker fortsätta i den riktning som det finns störst befolkningstäthet.

"Branch goals" är den externa funktion som avgör om en förgreningspunkt ska låta en ny gata skjuta ut från den, och i så fall i vilken riktning den nya gatan ska gå. Funktionen jämför befolkningstäthetsgradienten i fortsatta riktningen med gradienten i förgreningsriktningen, och om denna inte är alltför mycket lägre, så tillåts att en ny gata dras. När den nya förgrening dras görs även detta med hänsyn till var den högsta befolkningstätheten är belägen.

Det finns dock en yttre begränsning som påverkar de två ovanstående externa funktionerna, nämligen de användardefinierade parametrarna. Om dessa parametrar säger att en ny väg bara får svänga max 5 grader, så måste de externa funktionerna rätta sig efter detta även ifall de kanske hade föredragit en kraftigare sväng. Samma sak gäller för förgreningar, d.v.s. om parametrar säger att en ny förgrening måste vara vinkelrät mot huvudgatan, så kommer den att

dras vinkelrät oavsett vad "Branch goals " säger, förutsatt att det finns en tillräcklig population i den riktningen.

"Local constraints" kommer in när "Branch goals" eller "Cont goals" har bestämt sig för var dessa vill placera en gata. Ett antal tester gör för att se om gatan får placeras som tänkt, om den måste justeras något, eller om det helt enkelt är förbjudet att placera ett nytt vägsegment där. Nedan följer en uppräkning och förklaring av vilka kontroller "Local Constraints" gör efter det att den fått information om var "Branch goals" eller "Cont goals" vill placera ett nytt vägsegment:

- **Water-check:** Först kontrolleras om det nya vägsegmentet har placerats i vatten, och om detta är fallet försöker "Local constraints" vrida vägsegmentet så att det går längs med vattenkanten istället. Skulle det vara omöjligt att vrida vägsegmentet rätt så raderas det istället.
- **Elevation-check:** Även höjdskillnaden mellan vägsegmentets start- och slutpunkt kontrolleras för att se till att inga vägar går rakt upp för berg eller rakt ner i djupa dalar.
- **Cross-checking:** Om vägsegmentet korsar en annan gata så kopplas dessa samman. Detta för att undvika onaturliga korsningar.
- **Branchpoint-checking:** Om slutet på vägsegmentet är väldigt nära ett annat vägsegments förgreningspunkter så kopplas det aktuella vägsegmentet ihop med det närliggande vägsegmentets förgreningspunkt. Även denna check görs för att få en mer naturlig sammankoppling av gator.
- **Outside-of-map-check:** Vägar tillåts inte utanför gatan, och om de försöker läggas här, så tas de omedelbart bort.

Ovanstående regler gäller både för motorvägar och gator, men det finns ytterliggare en regel som bara gäller för gator:

- **Inside-populated-area-check:** Gator, till skillnad från motorvägar, får bara finns i områden med population. Om de ovanstående externa funktionerna försöker placera en gata utanför detta område, så kommer denna kontroll att ta bort den.

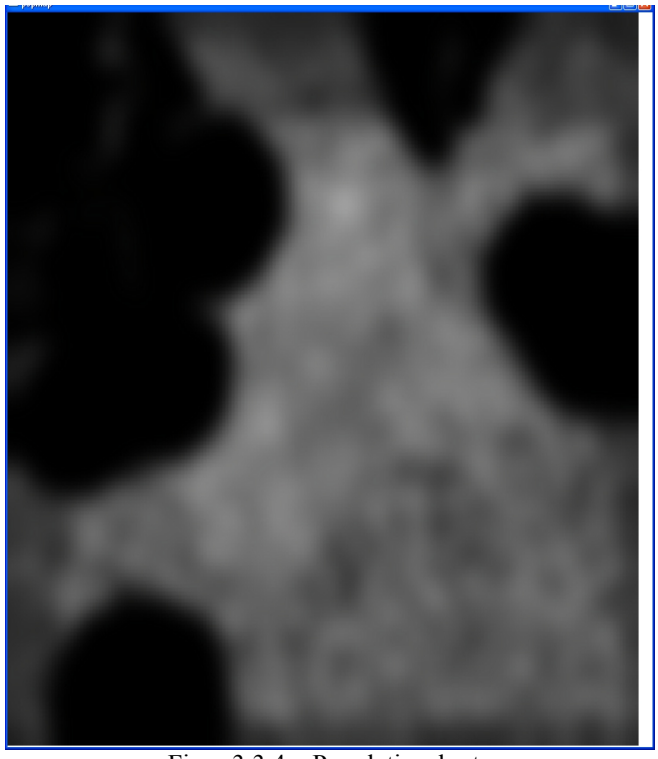
Det som skiljer Parish och Müllers system från ProCity Engine är alltså att istället för att bara ha "Global goals" så har ProCity Engine skilda externa funktioner för hur en väg ska fortsätta och för hur den ska förgrena sig. Detta i sig gör ProCity Engine betydligt mer uttryckskraftig och flexibel eftersom det finns betydligt fler möjligheter när det gäller designen av vägnätet. Givitvis är det så att Parish och Müllers design har betydligt fler valmöjligheter för tillfället, men potentialen i ProCity Engine är betydligt större. De möjligheter som användaren har, med användardefinierade parametrar, och möjligheter att välja de regler som passar, både för förgrening och fortsättning av vägar, gör att nästan alla önskade vägnät kan genereras, om bara speciella beteenden för vägarna beskrivs för systemet.

3.3.4 Exempel på vägnätsgenerering

Nedan följer en exemplifiering av hur ett vägnät genereras utifrån vissa indata, och vilket resultat som slutligen produceras, samt hur vissa av de kartor som användaren aldrig ser, men som programmet använder sig av, ser ut.

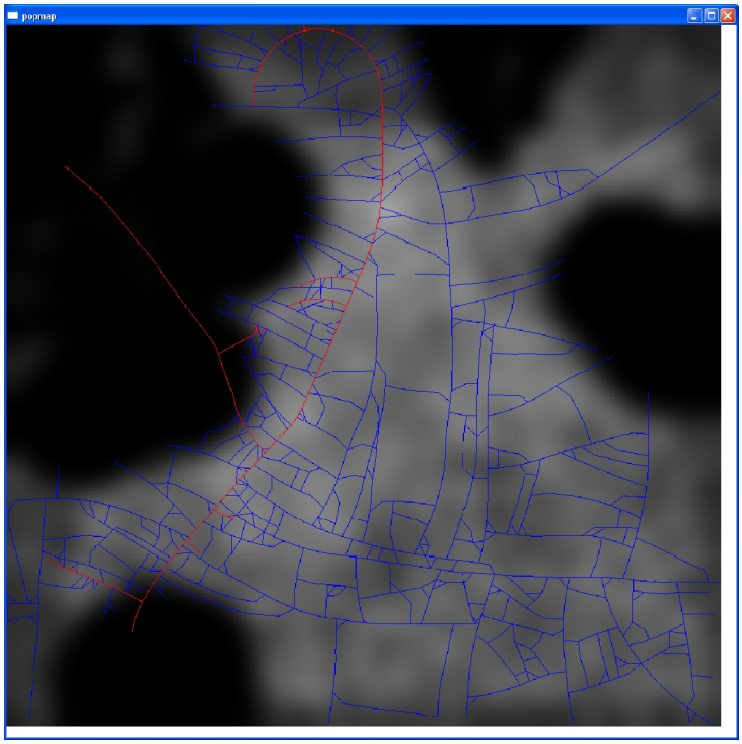
Figur 3.3.3 och figur 3.3.4 representerar den indata som RoadGen behöver, bortsett från de användaredefinierade parametrarna. Dessa två är inte gjorda av ProCity Engine, utan handgjorda för att enklare kunna visa några av de funktioner som RoadGen innehar.

Figur 3.3.3 – Höjdkarta

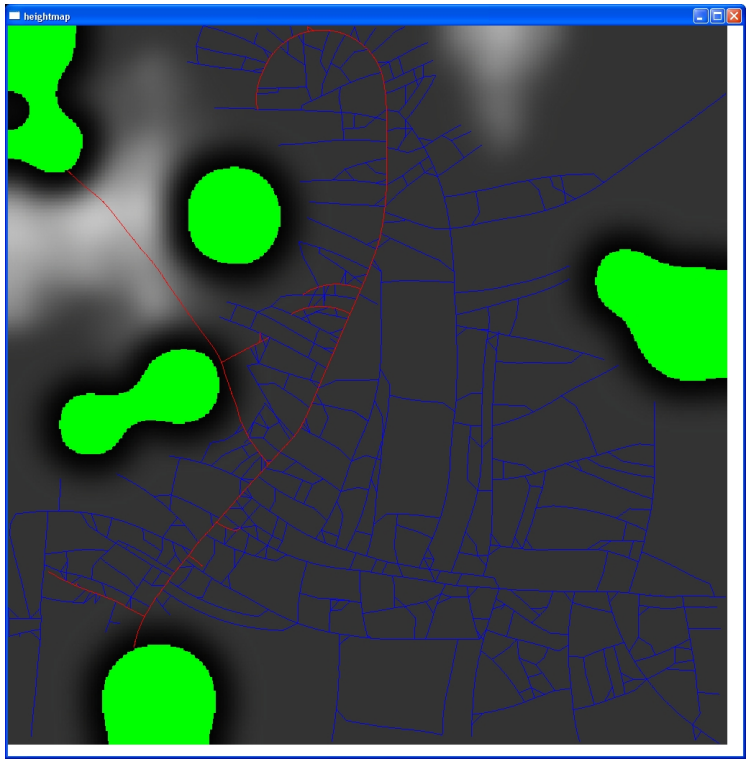


Figur 3.3.4 – Populationskarta

Vidare följer två bilder (figur 3.3.5 och 3.3.6) som visar hur vägnätet ser ut, överlagrat på höjd- respektive populationskarta. Detta för att visa dels hur vägarna undviker både vatten och berg, hur motorvägarna letar efter riktningar med stora populationer, samt att gator inte placeras utanför populationsområden.

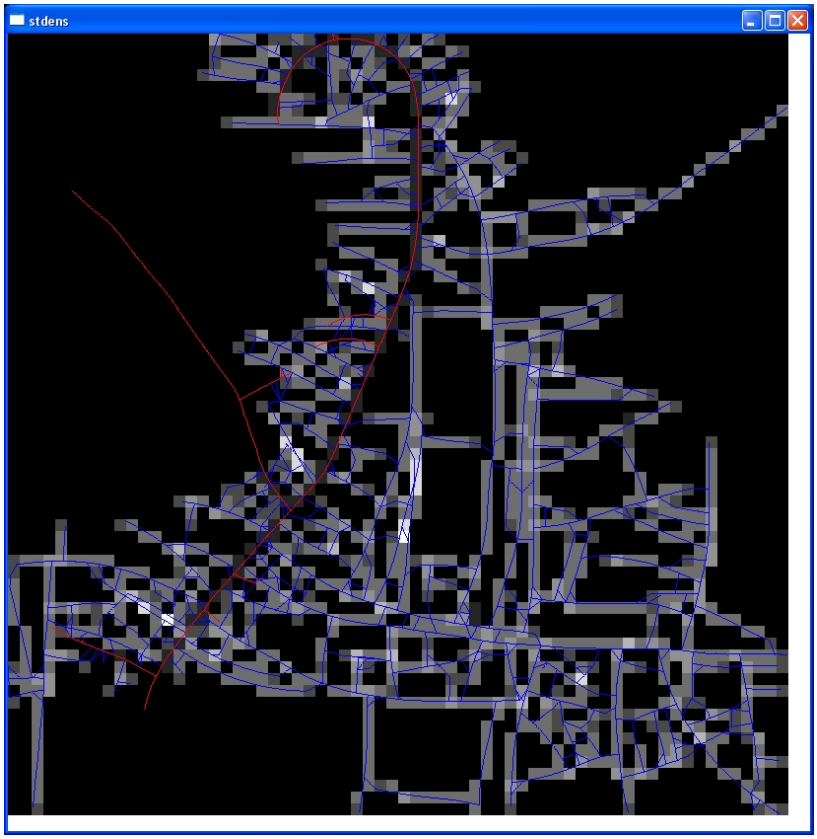


Figur 3.3.5 – Populationskarta med överlagrat vägnät

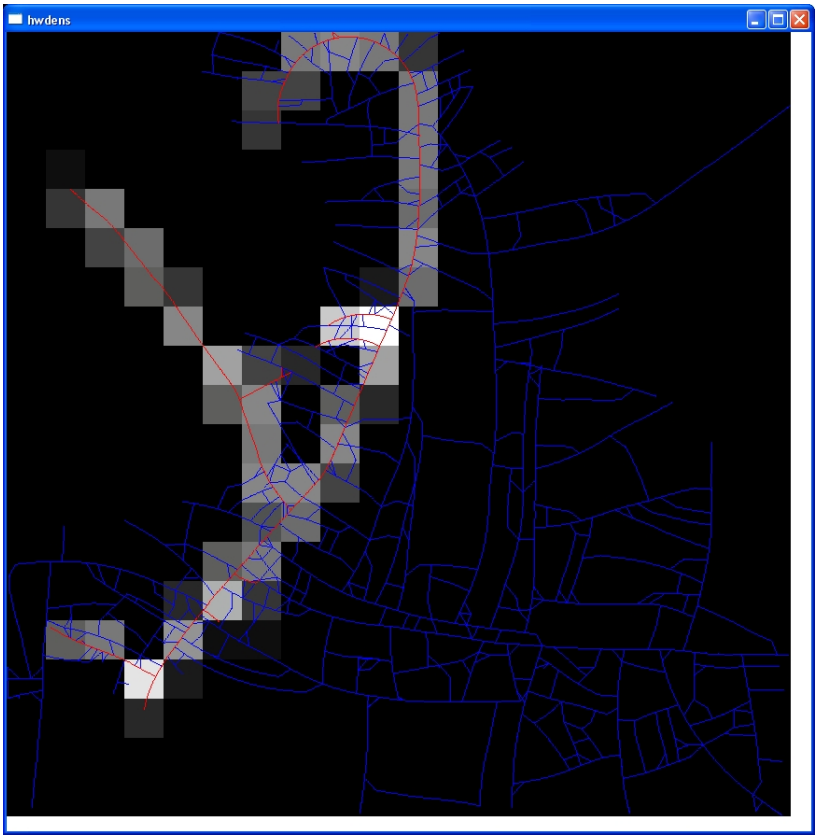


Figur 3.3.6 – Höjdkarta med överlagrat vägnät

Nedan följer två kartor (figur 3.3.7 och 3.3.8) som användaren normalt aldrig får se utan som bara används av programmet. Även dessa kartor är överlagrade med vägnätet för att sambandet mellan dessa enkelt skall kunna urskiljas. Dessa två kartor visar vägdensiteten och används som beskrivits ovan för att veta om mängden gator i ett område är för stort eller ifall det fortfarande går att placera ut fler gator.

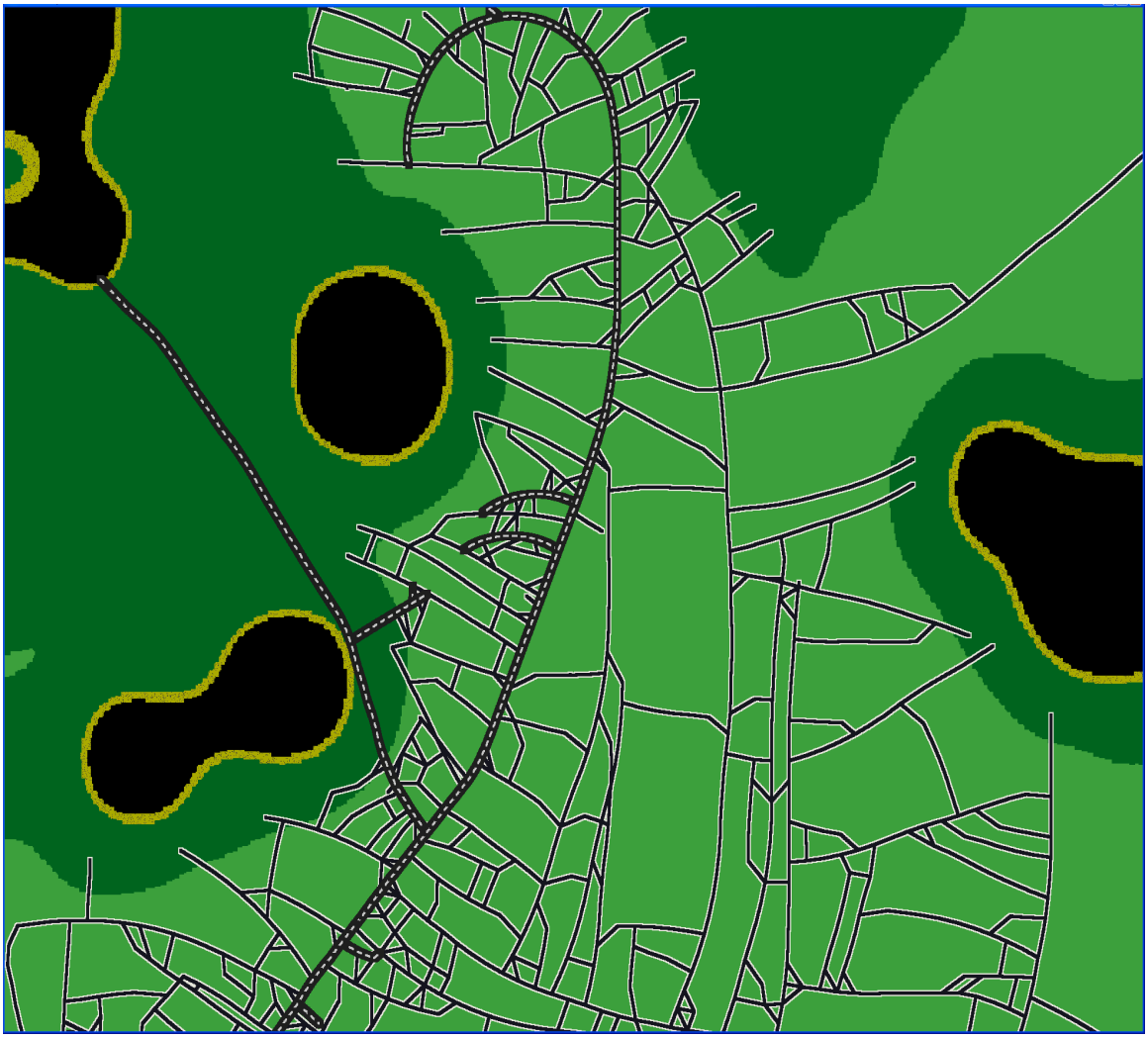


Figur 3.3.7 – Gatudensiteten



Figur 3.3.8 – Motorvägsdensiteten

Slutligen visas den textur som är det slutresultat som RoadGen producerat och som skall anges som textur till terrängmeshen i skriptfilen av renderer och sedan visas i Crystal Space.



Figur 3.3.9 – Färdig textur

3.4 LotGen

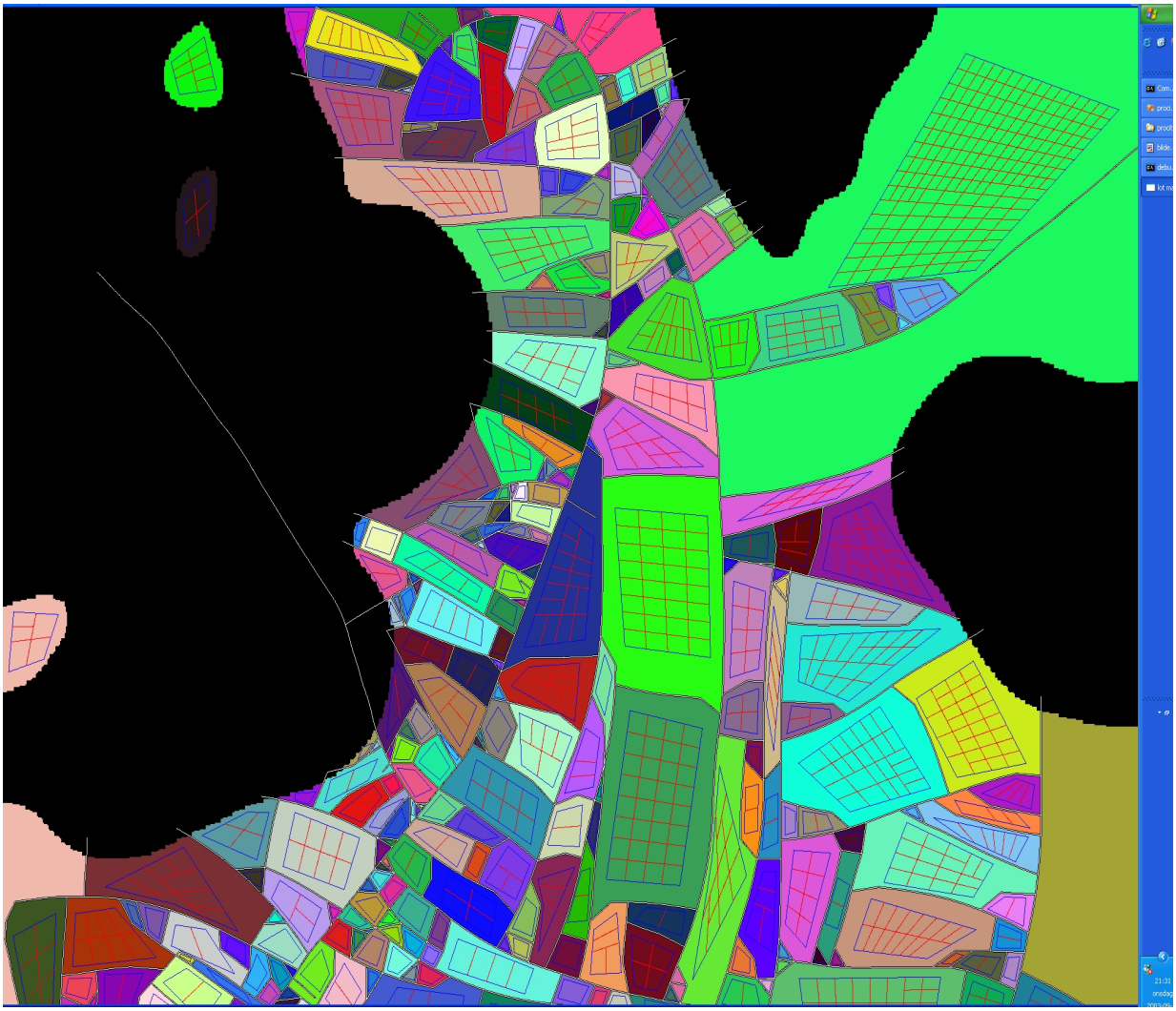
Innan byggnader kan placeras ut på det nyligen genererade vägnätet måste kartan delas in i någonting som kallas ”blocks”, vilka sedan styckas upp i så kallade ”lots” som var och en kommer att rymma en byggnad. Tillvägagångssättet är sådant att en fyrhörning dras inom varje inneslutet område, och dessa fyrhörningar blir det som kallas ”blocks”. Skapandet av dessa går till så att fyra punkter slumpas ut i olika riktningar från mittpunkten, och dessa punkter befinner sig alltid ett visst avstånd från kanten av inneslutningen. Därefter flyttas dessa punkter i alla möjliga kombinationer, alltid med ett visst avstånd från kanten, men med olika vinklar från mittpunkten, tills den största möjliga rektangulära formen har hittats. En inneslutning kan bestå av vägar, men även kanten på kartan, sjöar, eller kraftiga ökningar i höjdledd.

När dessa ”blocks” är definierade delas de in ytterligare genom att de två längsta sidorna delas på mitten varefter ett streck dras mellan dessa punkter, varefter samma process sker rekursivt på de två nya rektanglarna, och sedan på de fyra nya rektanglar o.s.v. När storleken på dessa indelade rektanglar når ett visst lägsta tröskelvärde kommer de inte längre att delas och kallas då istället för ”lots”. För att indelningen inte alltid ska bilda samma rutmönster är det möjligt att ha flytande tröskelvärden på storleken för ”lots”, så att alla inte blir lika stora, men det går även att i indelningen låta bli att dela rektanglarna på mitten och istället låta ett slumpvärde flytta dessa punkter åt olika håll och därmed få olika stora och olika formade rektanglar. Slutligen går det också att ställa in så att vissa ”lots” slumpmässigt tas bort, och dessa värden kan ställas in separat, dels för sådana lots som har kontakt med gatan, dels för de som inte har det. Nedan följer en beskrivning av det enskilda parametrarna och deras funktion.

- **Block_min_size**: Den minsta storleken på en inneslutning som tillåts för att ett block ska dras inom denna.
- **Block_max_size**: Den största storleken på en inneslutning som tillåts för att ett block ska dras inom denna.
- **Block_scan_accuracy**: När de fyra punkterna flyttas för att finna den optimala storleken för ”blocks” är det denna parameter som avgör hur noggrant detta sker, d.v.s. med hur små steg punkterna flyttas. Att minska denna gör att tiden för processen ökar dramatiskt eftersom processen går igenom alla möjliga kombinationer av de fyra punkterna.
- **Block_line_factor**: Den faktor som avgör hur långt från inneslutningens kant som de fyra punkterna befinner sig på, om man drar en rak linje mellan kanten och mittpunkten. Det som slumpas ut när det gäller de fyra punkterna är vilket gradtal de befinner sig på, och sedan hamnar de alltid på ett visst avstånd från mitten som är beroende på det totala avståndet i den riktningen, från mitten till inneslutningens kant, och Block_line_factor.
- **lot_min_area**: Det tröskelvärde som anger när en rektangel inte ska delas mer utan får lov att kalla sig för ”lot”.
- **lot_area_disruption:** Om detta värde är skilt från noll så är detta en varians till lot_min_area, vilket gör att alla lots får olika storlekar, inom vissa ramar.
- **lot_slice_disruption:** Denna faktor gör att indelningen av rektanglarna inte alltid blir ett strikt rutnät utan gör att indelningen som i vanliga fall sker från mitten av sidorna nu kan variera något, vilket kan resultera i olika stora ”lots”, men även ”lots” med olika rektangulära former.

- **lot_close_survival_ratio:** Den faktor som avgör hur vanligt det är för ”lots” i närheten av gator att inte slumpmässigt bli borttagna. Om denna står på ett, kommer alla ”lots” nära gator att finnas kvar.
- **lot_far_survival_ratio:** Den faktor som avgör hur vanligt det är för ”lots” som inte är i närheten av gator att slumpmässigt bli borttagna. Om denna står på noll, försvinner alla ”lots” som inte är i närheten av gator.

Nedan följer figur 3.4.1 som visar vilket resultat LotGen producerar. Alla ”blocks” har fått en färg slumpmässigt tilldelad för att på ett förtydligat sätt avgränsa dem från varandra. De blå strecken visar gränserna för ”blocks” medan de röda strecken är gränserna för ”lots”.



Figur 3.4.1 – LotGen

3.5 BuildGen

BuildGen är det pipelinesteg som ansvarar för generering av byggnader. Under detta pipelinesteg ska geometrin för samtliga byggnader i staden genereras. Förutom själva geometrin måste ett antal andra frågor besvaras för varje byggnad, dessa är bland annat - vilken typ har varje sida hos bygganden (eller rättare sagt: varje synlig sida), och vilken textur som ska mappas på denna byggnadssida. Vidare gäller att transformationen från byggnadens eget koordinatsystem, till hela stadens koordinatysystem måste bestämmas och sparas undan tillsammans med all annan information relaterad till en specifik byggnad samt för varje enskild byggnad.

I huvudsak utgör det ovanstående all funktionalitet som Procity för tillfället tillhandahåller med avseende på byggnader. Listan över olika möjliga aspekter av byggnadsgenereringen som skulle kunna ingå i examensarbetet, men inte gör det, är självfallet mycket länge; detta är dock en senare diskussion.

3.5.1 Grundläggande begrepp

För generering av byggnader använder sig BuildGen av stokastiska L-system där varje modul tolkas som en geometrisk operation på primitiva, 3-dimensionella objekt, i detta fall kuber. Dessa system, är liksom sina släktningar från roadgen, utökade och har dessutom externa funktioner för parametertillsättningar och parametermodifikation.Viktigt att påpeka är däremot att L-systemen i detta fallet intw överhuvudtaget har några självkänsliga egenskaper, ty ingen självkänslighet behövs i den modell som tillämpas för byggnadsgenerering i examensarbetet.

En byggnad kan för närvarande genereras i tre olika stilar. Dessa är:

1. En liten byggnad där en eller högst några familjer bor (bostadshus).
2. En större byggnad för kommerciella ändamål, eller en samling hyreslägenheter.
3. En mycket stor och dyr fastighet (skyskrapa).

Var respektive typ förekommer och varför har egentligen inte så mycket att göra med genereringen av en enskild byggnad. För enkelhetens skull dock, hanteras även denna problematik av buildgen i examensarbetet. Modellen som används för detta är väldigt enkel; man avgör byggnadens stil och övriga parametrar som höjd och horisontell utsträckning, på grundval utav byggnadens läge i förhållande till stadens centrum och i förhållande till befolkningstätheten där byggnaden ska sättas ut. Denna modell har naturligtvis inte någon överdriven realism till syfte, utan ska enbart vara enkel och tillräcklig för att demonstrera programmets potential och skapa tillfredsställande visuella effekter.

3.5.2 Parametrar

BuilGen, precis som allting annat i ProCity, styrs av användarangivna parametrar som beskriver (kontrollerar) nästan varje aspekt av byggnadsgenereringen som överhuvudtaget går att påverka. Låt oss titta närmare på vilka dessa parametrar är och vad de betyder. Paramerarna är följande:

bd_max_level
bd_max_scale_factor
bd_min_scale_factor
bd_min_split
bd_max_split

bd_initial_scaledown_factor
bd_max_height
bd_min_height
bd_height_variation

En i sammanhanget viktig kommentar är att det finns tre uppsättningar av ovanstående parametrar; en egen uppsättning för varje stil, dvs. för varje byggnadstyp. (Detta följes konsekvent även av det faktum att det finns tre olika L-system, ett för varje typ.)

Parametrarnas betydelse förklaras nedan. Märk väl att det inte går att förstå dess innebörd fullt ut innan läsaren stiftar bekantskap med övriga aspekter av buildgen-steget.

bd_max_level
Denna parameter anger den högsta detaljnivån som en byggnad genereras med. Mer konkret innbär det högsta antalet omskrivningar som L-systemet får lov att utföra på en modul som ingår i L-systemet.

bd_max_scale_factor
En faktor som anger hur mycket en modul (som ju representerar ett geometriskt primitiv) maximalt får skalas längs en godtycklig halvaxel. Och som bekant så kan en godtycklig punkt i det tredimensionella rummet beskrivas med hjälp av tre koordinater där varje koordinat hör till en av sex möjliga halvaxlar. Detta är kanske en något udda beskrivning av punkter i det tredimensionella rummet, men rent datatekniskt passar denna beskrivning mycket bättre för ändamålet ifråga, än den sedvanliga uppdelningen av R^3, i tre axlar. Rent matematiskt är det dock knappast någon skillnad, eftersom en utsaga som avser koordinater i det ena systemet alltid kan göras om till en fysikalisk ekvivalent utsaga som avser koordinater i det andra systemet.

bd_min_scale_factor
Som ovan fast parametern anger den minimala skalningsfaktorn längs en av sex möjliga halvaxlar.

bd_min_split
Denna parameter har en viss inverkan på hur en geometriskt primitiv delas upp i två stycken. En sådan uppdelning är den direkt orsaken till att det överhuvudtaget är möjligt att successivt förfina detaljnivån. Mer konkret gäller att parametern anger den minsta vid en uppdelning tillåtna kvoten: "den första mindre delens utsträckning" / "hela primitivets utsträckning innan uppdelningen"

bd_max_split
Samma som ovan fast kvoten som anges är den största tillåtna vid en uppdelning.

bd_initial_scaledown_factor
Denna parameter anger hur mycket det ursprungliga primitivet skalas ner (längs samtliga horisontella halvaxlar) innan L-systemet får börja operera på det. En noggrann läsare kommer förhoppningsvis fram till följande slutsats vid vidare läsning: Ju mindre denna parameter är, dessto större frihet kommer L-systemet att ha vid sina modifikationer av de ingående primitiven.

bd_max_height
Den största höjd en byggnad av en viss typ kan ha.

bd_min_height
Den minsta höjd en byggnad av en viss typ kan ha.

bd_height_variation
I builgen ingår en funktion som föreslår höjden på en byggnad innan den börjar genereras. Parametern ovan, anger variationen i denna föreslagna höjd, och anges relativt en enhet (dvs. talet 1).

Ovanstående var en kort förklaring av parametrarnas betydelse. Låt oss återigen understryka att det inte är meningen att dessa parametrar ska vara fullt förståeliga utan att man besitter ytterligare kunskap om buildgens funktionalitet. Denna kunskap ansamlas lämpligen genom vidare läsning av rapporten.

3.5.3 Datastrukturer

Indatat som används i buildgen för byggnadsgenerering är följande:

- en vektor med så kallade *lots*, där varje lot är ett område på marken där en byggnad kan tänkas (skall) stå. För mer information se kapitel om LotGen.

- populationskartan

- för övrigt ingår ett antal parametrar såsom stadens storlek, lotens läge i staden osv. i det som kan anses vara indata till BuildGen

Resultatet efter att detta pipeline-steget har utförts är en vektor med element av typen *mesh_class*. Vaje sådan instans utgör en byggnad. I datastrukturen ingår information om byggnadens geometri, vilka texturer som ska mappas på vilken sida och vilken transformation som ska användas för att placera ut byggnaden i staden. Detta är naturligtvis inte något slutresultat som kan erhållas ur ProCity, utan utgör indata till det pipelinesteg som kallas renderare och följer efter BuildGen (se kap. om renderare).

Mest intressant då man diskutetar BuildGens datastrukturer är kanske de interna datastrukturerna som BuildGen använder sig av för att generera en specifik byggnad. För det första ingår en klass med namnet: *cubeattr*, som är en subklass till *argument_class*. Objekt av typen *cubeattr* ingår som argument (/attribut) hos moduler i de L-system som används av BuildGen. Varje sådan instans beskriver ett geometriskt primitiv som i grund och botten är ett rätblock. Det som då behöver beskrivas är rätblockets storlek, och läge i förhållande till origo i det system som används för att beskriva hela modellen för en byggnad. Det finns dock ett avsprång från denna enkelhet som måste nämnas. För ett rätblock i BuildGens modell, gäller att en av de övre fyra kanterna kan förskjutas upp eller ner, varvid gäller att rätblocket efter en sådan förskjutning inte längre är ett rätblock. Denna egenskap har lagts till för att kunna generera sneda tak (till exempel på en villa) och andra geometriska strukturer som inte lätt kan brytas ner i rätblock enbart.

Ytterligare datastrukturer representerar de funktioner som avgör byggnadens typ och rekommenderad höjd. Dessa utgör en väldigt enkel modell för någonting som förtjänar en betydligt mer avancerad beskrivning för att ge rättvisa åt dess tämligen (för en teknolog) svåranalyserade egenskaper. Staden kan sägas undergå en tredelning i: utkanter, innerstad och centrum. De olika byggnadstyperna återfinns uteslutande i respektive område, dvs. bostadshus finns enbart i periferin, komerciella hus enbart i innerstaden och skyskrapor enbart i centrum. Denna modell är som sagt långt ifrån perfekt i någon mening. Dess stora fördel är givetvis den uppenbara enkelheten. I diskussionen i slutet av rapporten kommer skäl till att en såpass enkel modell har valts att beröras. Denna modell uppfyller dock sitt syfte och demonstrerar mer än väl programmets potential.

3.5.4 Stokastiska L-system, en kort repetition

Låt oss rekapitulera vad som sagts om stokastiska L-system. Ett stokastiskt L-system är ett L-system där varje regel utförs enbart med en viss sannolikhet i förhållande till andra regler som skulle kunna appliceras i ett givet sammanhang.
Ett stokastiskt utökat L-system är ett utökat L-system som även är stokastiskt.

3.5.5 L-system i BuildGen

BuildGen använder sig av stokastiska utökade L-system för generering av byggnader. Precis som i fallet RoadGen, har L-systemen i BuildGen externa funktioner för parametertillsättning samt parametermodifikation. De externa funktionerna beskrivs kortfattat i nästa stycke.

Vad L-systemen beträffar, gäller att det finns ett L-system för respektive byggnadstyp. Samtliga dessa L-system är dock uppbyggda enligt samma enkla modell med tre produktionsregler (modellen innefattar tre produktionsregler, vilket inte betyder att respektive L-system också innefattar just tre produktionsregler).
Denna allmänna mall för hur ett L-system i BuildGen är uppbyggt följer nedan:

regel 1: C(lvl , @attr) : lvl < max_level -> C(lvl + 1 , @attr1) C(lvl + 1, @attr2) : 0.4

regel 2: C(lvl , @attr) : lvl < max_level -> C(lvl + 1, @new_attr) : 0.4

regel 3: C(lvl , attr) : lvl < max_level -> C(max_level, attr) : 0.2

Detta var alltså reglerna. Låt oss förklara var och en av dem närmare. För det första gäller att modulen C har två parametrar. Den första parametern, *level*, anger vilken detaljnivå modulen befinner sig på. Ju högre värde, dessto mindre detalj utgör denna modul. Är detta värde lika med noll, betyder det till exempel att hela bygganden består av just denna modul, och detaljnivån är precis minimal.

Den andra parameten är av typen *cubeattr* som nämnts tidigare, och i detalj beskriver denna den geometriska struktur som en given modul motsvarar.

regel 1: C(lvl , attr) : lvl < max_level -> C(lvl + 1 , attr1) C(lvl + 1,attr2)

En förgreningsregel (branch). Här kan man se att en modul delas upp i två - ett rätblock delas in i två mindre. De externa parametrarna i denna regel läses och sättes av funktionen *branch()* , som registreras i vanlig ordning föreskriven i det i examensarbetet utvecklade API:et för L-system med externa funktioner (se kapitel 3.3.3 för mer information om externa funktioner, där det förklaras bättre i samband med de produktionsregler som används av RoadGen)

regel 2: C(lvl , @attr) : lvl < max_level -> C(lvl + 1, @new_attr) : 0.4

En regel för skalning. Likaså här finns det externa in- och utparametrar. Denna regel leder till anrop av funktionen *scale()*.

regel 3: C(lvl , attr) : lvl < max_level -> C(max_level, attr) : 0.2

Äntligen en regel utan externa funktionsanrop; skulle kunna kallas för en *finalize-regel*, ty dess enda syfte är att med viss sannolikhet förhindra ytterligare förfining av detaljnivån hos ett primitiv ifall sådan förfining fortfarande har chans att ske. Genom att betrakta regeln ser man snabbt att det enda som görs är att den parameter som talar om vilken detaljnivå primitivet för tillfället befinner sig på, sätts till sitt maximala värde. I samtliga regler ingår det som prekvisitum att denna parameter ska vara mindre än sitt max-värde för att regeln ska kunna tillämpas. Således, om denna parameter är satt till sitt högsta möjliga värde, kommer inga ytterliggare produktionsregler att appliceras på denna modul.

3.5.6 Externa funktioner

De externa funktionerna, dvs. *branch()* och *scale()*, bestämmer parametrarna då en modul skall delas upp i två och då storleken eller formen på en modul skall ändras. Båda fallens beskaffenhet är ytterst beroende av stilen på den byggnad som genereras. Till exempel så skapas inga sneda tak på skyskrapor, men däremot så kan det göras på villor. Således kommer *scale()* aldrig att sänka eller höja en övre kant hos en kub som ingår i en skyskrapa.

I dessa funktioner finns det hårkodad funktionalitet för någonting som egentligen skulle vara betydligt mer parameterstyrt, dvs. naturen hos de olika transformationer som kan utföras på beståndsdelar av en byggnad. Tilläggas skall dock att sådan fast bunden funktionalitet lätt kan flyttas ut i form av parametrar. Anledningen till att detta inte har gjorts är att detta examensarbete genomförts av personer med mycket föga kunskaper för estetik i allmänhet och arkitektur i synnerhet. Därför är det ytterst svårt att på ett bra sätt dela upp en byggnad i grundläggande beståndsdelar som sedan kan modifieras för att erhålla stor variation bland de byggnader som genereras.

Tillvägagångssättet vid framställningen av dessa funktioner har varit sådan att olika möjliga transformationsvarianter testades för respektive stil, och när resultat som i alla fall till synes är tillfredsställande har erhållits, var det ett tecken på att funktionerna inte skulle ändras mer.

3.5.7 Komplexitetsanalys

Definiera följande:

B = antalet byggander
D = detjaljnivån hos varje byggnad
K = tidskomplexitet hos BuildGen steget

då gäller att antalet moduler efter D iterationssteg är 1.5 ^ D

Och eftersom det i snitt finns hälften så många moduler per iterationssteg och man utför D iterationssteg per byggnad, gäller att

K1(D) = O(D * (1.5^D) / 2)

är tidskomplexiteten för en byggnad. Den totala tidskomplexiteten blir då:

K(D, B) = O(B * D * (1.5^D) / 2)

eftersom det finns B byggnader att generera. Eftersom D oftast är ett litet tal, kan man säga att tidskomplexiteten är mer eller mindre linjär i antalet byggnader, dvs. O(B).

3.5.8 BuildGen ur oberoende- och realtidsaspekt

Att generera en byggand går väldigt fort, och således är det inga problem att generera byggnader i realtid efterhand som man behöver dem. Inte heller borde oberoende generering av byggnader ställa till med problem i sig, ty givet parametrarna, så är ju en byggnad oberoende (ur samtliga genereringaspekter) av de andra.

Däremot gäller att för att ta fram parametrarna för en specifik byggnad i ProCity-modellen, behövs ett färdigt underliggande vägnät. Problemet att generera byggnader i realtid och oberoende av varandra faller således ner på problemet att göra samma sak med vägnätet i ProCity-modellen. Och detta är ju som bekant en helt annan femma.

3.6 TexGen

Denna del av genereringsprocessen skulle ursprungligen skapa slumpmässiga texturer procedurellt, vilket då skulle garantera nya texturer för varje ny stad. Detta visade sig dock alltför tidskrävande implementeringsmässigt och istället läses här alla texturer i en aktuell folder in så att de sedan kan användas vid skapandet av en skriptfil som kan läsas av Crystal Space.

För att texturerna ska läsas in och lagras i genereringsprocessen krävs att deras namn följer ett visst mönster. Detta mönster presenteras och förklaras i tabell 3.6.1 och exemplifieras nedan i tabell 3.6.2.

Tecken 1-3	”Tex”	Alla texturer börjar med dessa tre bokstäver.
Tecken 4-5	Två siffor	Vilken arkitektonisk stil texturen representerar.
Tecken 6-7	Två siffor	Vilken del av ett hus texturen ska täcka.
Tecken 8-9	Två siffor	Om tecken 4-7 är identiska för två texturer används dessa tecken för att skilja dem åt.
Tecken 10-13	”rx”, ”ry” eller ”nr”	rx = upprepningsbar i x-led ry = upprepningsbar i y-led nr = icke upprepnigsbar
Tecken 14-16	Tre Siffror	Texturstorlek i x-led.
Tecken 17-19	Tre Siffor	Texturstorlek i y-led.
Tecken 20-23	”.jpg”	Bildformatet.

Tabell 3.6.1 – Mönster för texturnamn

Tex010401rxry100100.jpg
Tex042002rxnr100001.jpg
Tex997643nrry100010.jpg

Tabell 3.6.2 – Exemplifiering av texturnamnsmönster

Anledningen till angivandet av arkitektoniska stilar är givetvis för att det ska vara möjligt att enkelt kunna få byggnader i ett visst område att ha samma typ av arkitektur, som t.ex. China Town i New York, eller Montematre i Paris, och på detta sätt få en trovärdigare stad. Andra indelningar kan givetvis vara att texturen tillhör industri, affärer eller bostadsområde men hur som helst så är meningen att en indelning ska kunna göras, kanske med hjälp av en karta liknande populationskartan, som delar in staden i olika delar som sedan använder olika arkitektoniska texturer.

Ett hus som ingår i en viss arkitektonisk stil måste ha ytterliggare en uppdelning för att dörren inte ska hamna på taket, och taket ut mot gatan, nämligen vilken sida av huset som texturen är menad för. Den uppdelning som är tänkt i ProCity Engine är sådan att taket har en viss textur, och texturer som vetter mot en gata har en viss textur, en typ av texturer har portar, en typ ax texturer är enfärgade, undersidor av delar av byggnader som inte har kontakt med marken har en textur, samt att alla andra sidor har en viss textur. För sammanfattning och vilket nummer som verkligen motsvarar vilken typ av textur, se tabell 3.6.3.

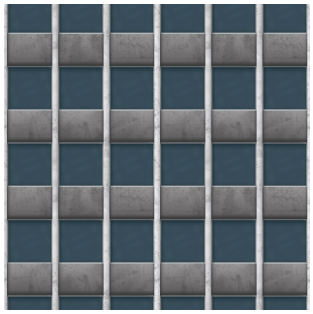
Street [0]	En fasad som är riktad mot gatan
Normal [1]	En fasad som inte är riktad mot gatan

Solid [2]	En helgfärgad textur som kan användas t.ex. på tak
Port [3]	En fasad med port
Bottom [4]	En textur som är riktad neråt mot marken
Roof [5]	En taktextur

Tabell 3.6.3 – Olika sidor på en byggnad

Slutligen måste det också finnas ett index som säger att det finns många olika sorters tak i en viss stadsdel så att inte alla hus inom ett visst område ser lika ut. Naturligtvis kommer husen inte att se geometriskt lika ut med bara en sorts textur, men alla väggar som vetter mot en väg kommer att se lika ut.

Anledningen till att ange om en textur är upprepningsbar i någon ledd är för att ProCity Engine ska veta om den ska upprepa eller sträcka ut texturen. Figur 3.6.1 nedan är ett exempel på en upprepningsbar textur, medan figur 3.6.2 är exempel på en som inte är detta.



Figur 3.6.1 – Upprepningsbar hustextur

Figur 3.6.2 – Icke upprepningsbar husfasad

Med texturstorlek menas hur stor texturen skulle kunna antagas vara i verkligenheten.. Figur 3.6.1 skulle till exempel kunna vara ungefär 12 meter i y-led eftersom den är fyra våningar, medan figur 3.6.2 kan antagas vara över 100 m. Detta är en bedömningsfråga som användaren måste ta ställning till när hon kategoriserar texturen. Anledningen till att dessa värden behövs är för att när renderingen sker så kommer texturen att sträckas ut för att passa de hus den ska sitta på, och detta betyder att t.ex. figur 3.6.1 kommer att upprepas sig ett visst antal gånger beroende på hur hög byggnaden med denna textur ska antagas vara. Vilket bildformat som används är upp till användaren, och har ingen större betydelse bortsett från kvalitet på bilden och andra liknande egenskaper.

3.7 Renderer

Den slutliga delen av genereringsprocessen tar del av all information som tidigare steg har producerat och konverterar denna till en skriptfil som Crystal Space kan läsa och rendera. I skripfilen finns information om allt som den genererade staden ska innehålla: hus, gator, terräng, ljuskällor osv., samt information om vilka texturer som finns tillgängliga och vilken utgångspunkt kameran har i staden. Nedan följer en exemplifiering och förklaring av hur en sådan skriptfil kan se ut.[b]

<world>
<textures>
//Här anges de texturer som ska användas.
</textures>
<materials>
//Här anges vilka texturer som ska bilda vilka material.
</materials>
<plugins>
//Här anges vilka plugins som används i skriptfilen.
</plugins>
<start>
//Här anges startpositionen för kameran.
</start>
<settings>
//Här anges globala valmöjligheter som t.ex. vilken ljusstyrka det ambienta ljuset har och hur stor en sida maximalt får vara för att den fortfarande ska ljussättas.
</settings>
<meshfact name = ****>
//Här anges om skriptfilen kommer att innehålla några färdiggjorda meshobjekt, som t.ex. Crystal Space egna terrängmesh. Det går även här att ange mallar till meshobjekt om det är uppenbart att flera meshobjekt av samma typ kommer att användas.
</meshfact>
<sectorname = ****>
//I varje sektor går det att lägga till ett önskat antal meshobjekt, men det går även att ange om det önskas dimma och liknande i den aktuella sektorn.
<meshobject name = ****>
// Ett meshobjekt kan vara ett träd, ett hus, en terräng eller något liknande. Det finns mängder av valmöjligheter att ställa in för varje enskilt meshobjekt.
</meshobject>
<light name = ****>
//Här anges de ljuskällor som ska finnas i sektorn.
</light>
</sector>
</world>

En stor del av det totala intrycket på hur staden ser ut är själva terrängen, vilken Crystal Space har goda möjligheter att modellera med sitt terräng-meshobjekt, som representerar den

terrängmotor som finns i spelmotorn. Grundprincipen med denna terrängmotor är att hela terrängen är indelad i block, vilket med standardinställningar betyder att den är indelad i 8x8 block, men detta antal kan ökas eller minskas efter behov. I detta projekt används bara ett block, eftersom endast en stor textur används för att representera terrängytan. Om flera texturer ska användas är det lämpligt att ha ett större antal block, eftersom texturerna då kan appliceras på individuella block.

Det finns ytterligare anledningar till att rekommendera användandet av ett större antal block. Den detalj nivå som ska visas beräknas per block, vilket gör att användandet av endast ett fåtal block hindrar olika delar av terrängytan från att visas i olika detaljnivå, även ifall två punkter på den befinner sig långt ifrån varandra. Det finns också begränsningar i hårdvara på så sätt att det inte går att ha hur stora texturer som helst i minnet, och då kan det vara lämpligt att dela upp dessa och sprida ut dem över flera block.

Vidare delas varje block in i ett rutmönster, som kan vara mer eller mindre komplext, och ju färre block som används, desto tätare rutnät måste användas för att plötsliga skiftningar i terrängen ska synas.

Andra inställningar för detta meshobjekt så som hur långt ifrån en textur kameran ska vara innan detaljnivån skiftar, vilken fil som innehåller höjdkartan, ambient ljus, och liknande ges även möjlighet att ändra på i skripfilen.

Nedan följer exempel på hur ett terräng-meshobjekt kan se ut, och det följs av en beskrivning på hur ett enkelt hus meshobjekt kan se ut.

---------- Terräng meshobjekt

```
<meshobj name="terrain">
        <plugin>terrfunc</plugin>
        <zuse />
        <params>
                <factory>terrFact</factory>
                <blocks x="64" y="64" />
                <correctseams w="64" h="64"/>
                <material>roadmap</material>
                <topleft x="0" y="0" z="0"/>
                <scale x="150" y="1" z="150" />
                <heightmap>
                        <image>heightmap.bmp</image>
                        <scale>0.5</scale>
                        <shift>0</shift>
                </heightmap>
         </params>
</meshobj>
```

---------- Hus meshobjekt

```
<meshobj name='Building 71'>
        <plugin>thing</plugin>
        <zuse />
        <params>
                <v x='30' y='21' z='420'/><v x='40' y='21' z='420'/>
                <v x='30' y='21' z='410'/><v x='40' y='21' z='410'/>
                <v x='30' y='31' z='420'/><v x='40' y='31' z='420'/>
                <v x='30' y='31' z='410'/><v x='40' y='31' z='410'/>
                <material>stone</material>
                <texlen>128</texlen>
                <p name='front'>
                        <v>5</v> <v>4</v> <v>0</v> <v>1</v>
                </p>
                <p name='back'>
                        <v>3</v> <v>2</v> <v>6</v> <v>7</v>
```

```
                    </p>
                    <p name='right'>
                                <v>1</v> <v>3</v> <v>7</v> <v>5</v>
                    </p>
                    <p name='left'>
                                <v>2</v> <v>0</v> <v>4</v> <v>6</v>
                    </p>
                    <p name='up'>
                                <v>4</v> <v>5</v> <v>7</v> <v>6</v>
                    </p>
                    <p name='down'>
                                <v>1</v> <v>0</v> <v>2</v> <v>3</v>
                    </p>
                    </params>
          </meshobj>
```

---

4 Avslutning

4.1 Resultat

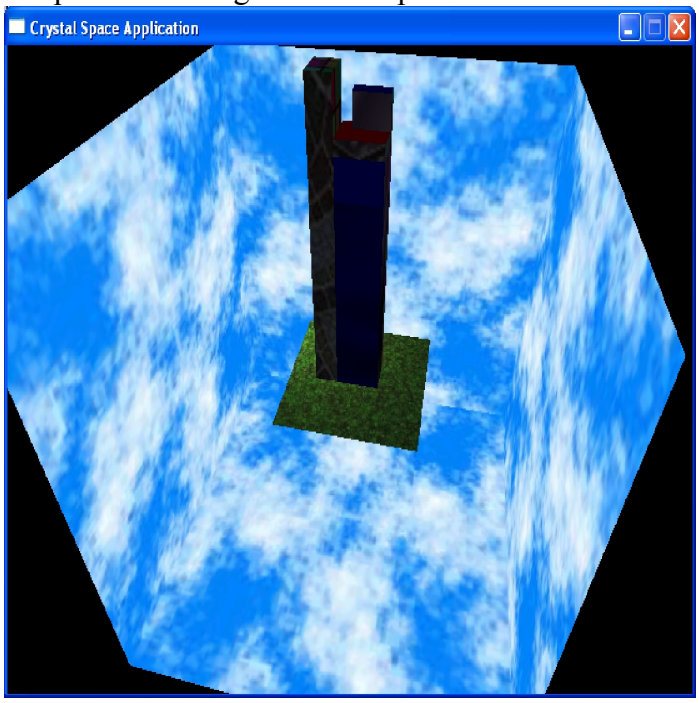
Följande kapitel kommer att visa de resultat som ProCity Engine producerar, till största delen i form av screenshots och medföljande förklaringar. Med vilka parametrar sceenshotsen har genererats kommer inte att tas upp, eftersom liknande resonemang redan förts i föregående kapitel, utan här kommer endast resultat att visas så att ProCity Engines potential verkligen åskådliggörs. Vägnäten och byggnaderna i nedanstående bilder är helt genererade av ProCity Engine, men vissa höjd- och populationskartor som använts kan vara förgenererade eftersom detta kan behövas för att visa speciella funktioner som ingår i genereringsprocessen.

De två första bilderna (figur 4.1.1, 4.1.2) visar översiktsbilder av staden, och detta för att ge ett helhetsintryck av vilken typ av stadskänsla som ProCity Engine kan generera. Det går i dessa bilder tydligt att urskilja att byggnaders form och deras texturer är bundna till deras plats i staden. Skyskrapor genereras inte i förorterna utan i de centrala delarna av staden, medan villaliknande byggnader placeras i de mindre folktäta delarna.

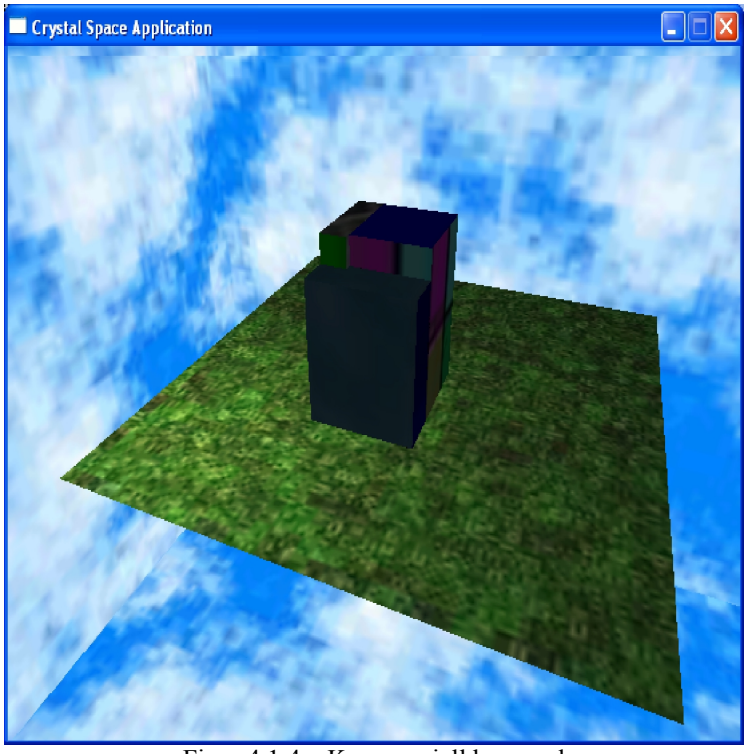
Figur 4.1.1 - Överblicksbild 1

Figur 4.1.2 - Överblicksbild 2

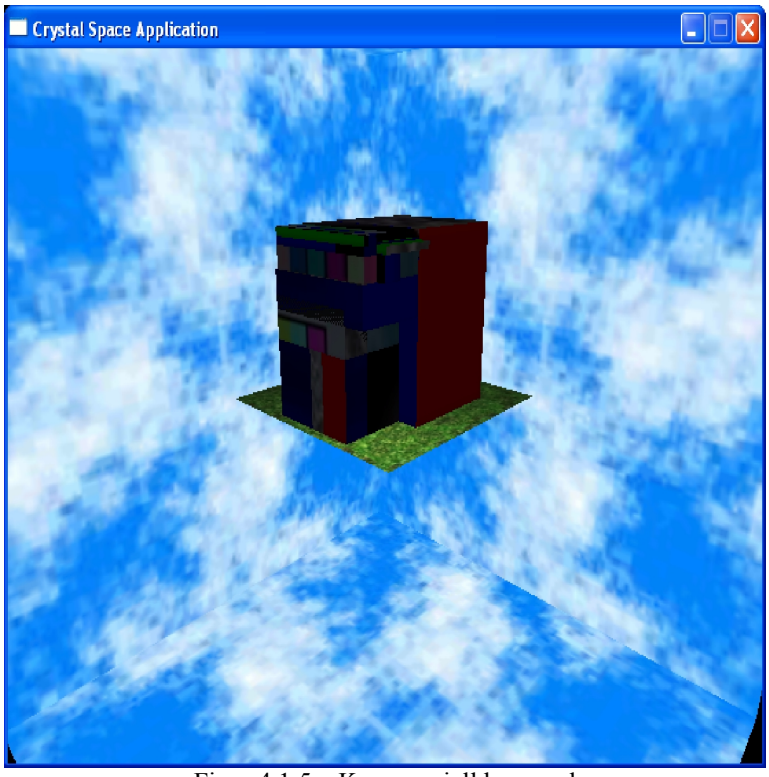
De tre följande figurerna (figur 4.1.3, 4.1.4, 4.1.5) visar närbilder på två olika typerna av byggnader och deras distinkta skillnader. Bilderna åskådliggör det resultat som BuildGen producerar och visar byggnader med betydligt högre detaljnivå än de som visades i överblicksbilderna ovan. Dock har dessa byggnader inga speciella texturer, utan varje sida har givits en viss färg. Förutom denna typ av byggnader är det även möjligt att skapa byggnader med sneda tak, vilka då på ett trovärdigt sätt kan representera t.ex. villor.



Figur 4.1.3 - Skyskrapa



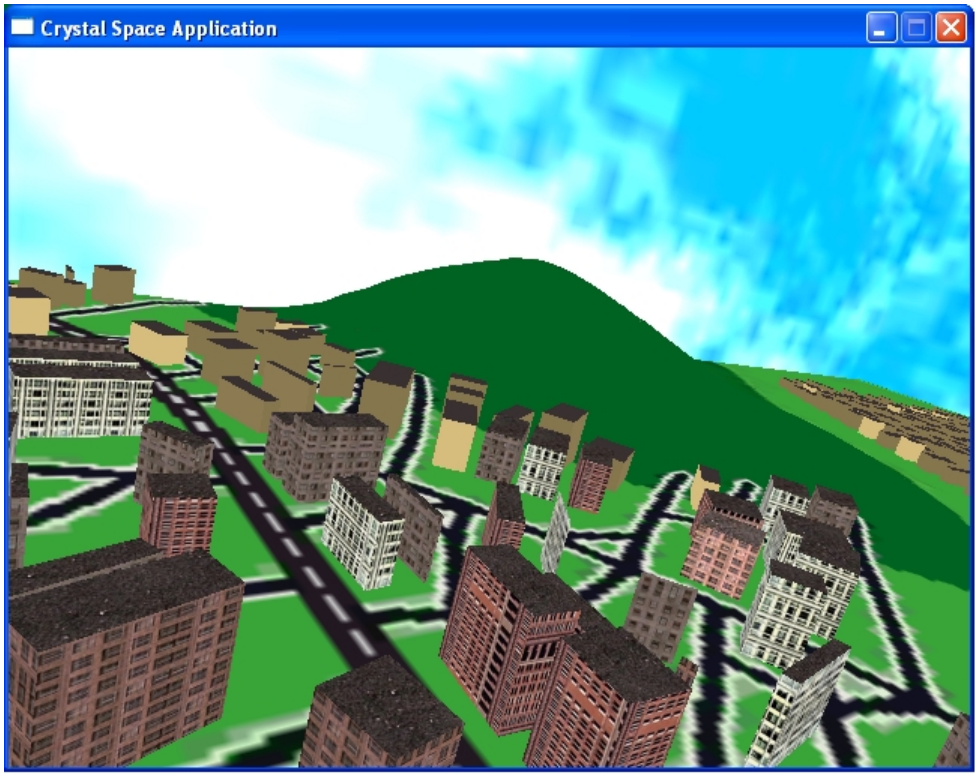
Figur 4.1.4 – Kommersiell byggnad



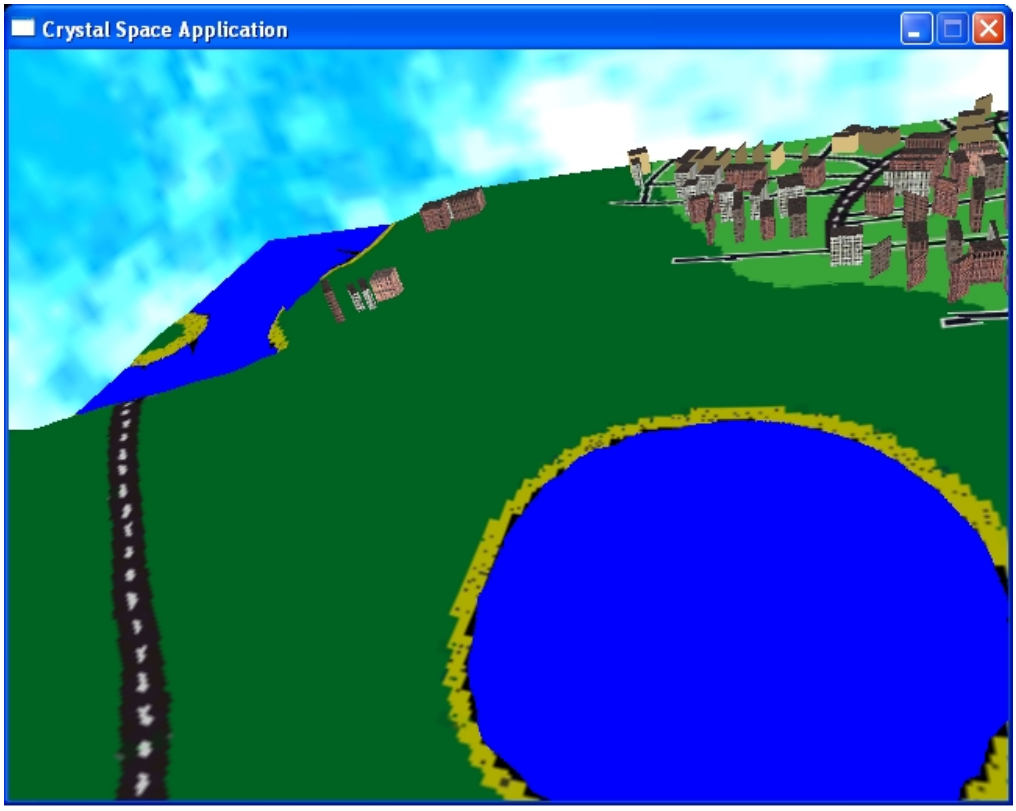
Figur 4.1.5 – Kommersiell byggnad

Kapitlet avslutas med att i figurerna nedan illustrera delar av terrängen, samt hur vägnätet ser ut efter att ha renderats av Crystal Space.

Figur 4.1.6 - Vägnät



Figur 4.1.7 – Byggnader med kulle i bakgrunden



Figur 4.1.8 – Sjöar

4.2 Utvecklingsmöjligheter

Följande kapitel beskriver vad i de olika delarna av ProCity Engine som kan förbättras och ger exempel på hur detta kan göras. Det finns naturligtvis mer saker att göra än som beskrivits här, eftersom detta projekt har oändlig potential, men exemplen här är de som är mest rimliga och inom de områden som angivits i projektbeskrivningen.

4.2.1 MapGen

Den kartgenerator som konstruerats har gjorts från grunden, utan speciella algoritmer, perlinbrus eller någon annan beprövad teori, och den stod inte beskriven i Parrish och Müllers rapport. Här finns goda möjligheter för förbättringar, eftersom det som gjorts är väldigt grundläggande. Både sjöar och berg kan på ett flertal sätt göras mer naturliga, utan de geometriskt rena former som de nu innehar, och landskapet kan göras betydligt mer varierat istället för en platt yta med höga toppar och djupa hål. Även ifall befolkningskartan är kopplad till höjdkartan så kan denna del göras betydligt bättre genom att göra processen mer känslig för populationstäthet i närheten av berg och sjöar. Ännu en gång gäller det att få bort den rena geometrin och få en mer flytande övergång.

En trevlig funktion hade varit möjligheten att göra kanaler, eller floder som ringlade sig igenom kartan. Detta hade också öppnat vägen för att skapa dynamiska vattenfall som då skulle uppstå när en flod gick i en riktning med alltför stor höjdgradient.

Ytterliggare en funktion hade varit att generera klimat kartor som Renderer sedan kunde använda för att placera ut regn, snö och dimma i vissa områden. Liknande resonemang gör det även intressant att skapa föroreningskartor, vilka lade grunden till smog och rök i de centrala delarna av staden.

4.2.2 RoadGen

En av de enklare utvecklingar som kan göras är att lägga till fler "Cont goals" och "Branch goals" för att ge en ökad möjlighet att uttrycka olika sorters gatunät. Givetvis gäller här också att sätta sig in i hur parametrarna fungerar och hitta den perfekta parameterkombinationen för det speciella gatunät som skall uttryckas. "Local constraints" går även att utveckla för att få mer precisa regler för var gator är tillåtna och var de inte är det.

Det kan vara möjligt att användaren har intresse av att bestämma att det ska finnas gator på vissa delar av kartan, och detta kan i princip redan göras genom att ange ett godtyckligt antal initialsegment, d.v.s. systemets axiomuttryck. På detta sätt kan användaren få gator att börja där hon vill, men problematiken ligger i att detta inte smälter väl ihop med vad systemet annars genererar i stadens övriga delar. För att få detta att fungera krävs nog väl övervägda parametermodifikationer.

Ytterliggare ett problem kan vara att användaren finner den textur som RoadGen producerar vara av alltför dålig upplösning. Det finns ett antal angreppssätt för att lösa detta problem. Det enklaste är att bara öka texturens pixelstorlek men problemet med detta när det gäller Crystal Space är att motorn bara tar in texturer av en viss största storlek, och om denna överstigs så trycks texturen ihop. Lösning blir då att skapa texturen i olika delar, vilket gör att upplösningen då kan ökas så mycket som önskas.

4.2.3 LotGen

Förutom att få en karta med alla "lots" kan detta steg även producera zonkartor som beskriver vilka områden som har vilka stilar. Vilka delar är industriområde, centrum, eller landsbyggd, så att speciella texturer hamnar på rätt typ av byggnader.

Om dynamiska människor implementeras så kan detta steg även ange vilka typer av människor som bor på olika ställen, och då ange vilka texturer som ska användas. Även kartor

som hade anknytning till barnafödande kunde vara intressanta för att då skapa mer barn-figurer i dessa områden. Högt barnafödande skulle t.ex. även kunna öka möjligheten att något hus i området fick texturer som gjorde att den liknade en skola.

Ytterliggare en karta som kan ge mer liv åt staden kan vara hur stor brandrisken är i ett visst område, och då göra att något enstaka hus i staden fick ytterliggare dynamiska effekter som gjorde att det såg ut som att det brann. Andra liknande kartor kan göras som avgör var och hur ofta olika dynamiska effekter ska visas.

Anledningen till att ovanstående förslag bör göras här och inte i MapGen är givetvis för att det bör tagas hänsyn till vägnätet.

4.2.4 BuildGen

Det kan vara intressant att kanske lägga till byggnader som användaren har gjort själv i t.ex. 3D Studio eller något liknande program. Förutsatt att användaren kan skripta sin egen byggnad så är det möjligt att gå in i den färdiga skriptfilen och byta ut någon befintlig byggnad med den som användaren själv har skapat. Lämpligt är då att hitta en byggnad som är lika stor eller större än användarens byggnad och ta dess plats, eftersom byggnaden då med mindre sannolikhet kommer att skära sig mot resten av staden. Det största problemet här förutom förutsättningen att användaren kan skriva sin egen skriptfil är att användaren måste vara väldigt försiktig så att hennes byggnad inte går rakt ut i en sjö eller över en väg, bara för att den "lot" som valdes var alltför liten. Ifall användaren vill ha sin egengenererade byggnad på en speciell plats så förutsätts dessutom att hon kan extrahera denna information från ProCity Engine, vilket i och för sig inte är omöjligt att möjliggöra.

Det skulle nog dock vara möjligt att efter det att kartan som visar alla "lots" genererats, visa denna för användaren och fråga om hon vill reservera någon "lot" för egna byggnader. Här skulle till och med kunna skapas en byggnadskonstruktor som lät användaren i programmet skapa en önskad byggnad på en önskad "lot". Detta kräver omfattande arbete, men är inte omöjligt att implementera.

För att ett hus ska se ut på ett visst sätt så måste ett val av parametrar göras. För att skapa vissa stilar måste dessa direkt kopplas till parametrar, och eftersom parametrarna i nuläget är väldigt svårförståeliga krävs här ett nära samarbete mellan en tekniskt kunnig och även en estetiskt sådan. Detta bör kunna förenklas till sådan grad ett endast ett estiskt kunnande skall räcka för att skapa nya intressanta stilar.

4.2.5 TexGen

Den mest uppenbara förbättringen kan vara att utveckla TexGen till det som beskrevs av Parish och Müller. Det är beskrivets relativt bra ovan, men det är inget trivialt arbete och kräver säkerligen ett flertal dagars arbete. När TexGen väl semi-procedurellt kan producera texturer finns det givetvis ytterliggare saker som kan utvecklas som att till exempel även producera texturer för träd, parker och liknande.

Om dynamiska människor läggs in i programmet kan denna del sköta skapandet och kategoriseringen av dessa. De procedurella skapandet av människotexturer kan dock föra vissa problem med sig, och är nog ett examensarbete i sig.

4.2.6 Renderer

En möjlig utveckling av denna del av ProCity Engine är att ge användaren möjlighet att välja vilket format hon vill ha skriptfilen på, men det kräver givetvis kunskap om exakt hur olika programs och spelmotorers skriptfiler fungerar och är uppbyggda.

Det finns ett flertal tillägg som här kan göras för att få staden att bli mer levande. Effekter så som snö, regn, smog, dimma och liknande kan läggas in, men även sådant som brand, och fyrverkerier. Det finns möjligheter att lägga in dynamiska träd, sol och måne som rör sig över

himlavalvet, trafikljus i gathörnen, bänkar i parker, vattenfall, och mycket annat som Crystal Space stödjer. Generering av gatunät och byggnader är en sak, men det är de små detaljerna som gör staden levande.

Angående marktexturen och att bara ha en enda stor sådan finns ytterliggare ett problem när det gäller rendering. Eftersom bara ett enda block används när man har en enda stor textur betyder detta att hela denna måste renderas när kameran är riktad mot någon del av den. Om marktexturen däremot delas upp i många olika små delar, och på så sätt även ger möjlighet att öka upplösningen på den, så kan dessa sättas fast var och en på olika block, vilket gör att det finns möjlighet att rendera terrängen med fler block än ett. Crystal Space är uppbyggt så att om kameran ser någon del av blocket så renderas hela blocket, men detta betyder att om terrängen är uppdelad i flera block behöver inte hela meshen renderas om kameran bara ser en liten del av den.

4.2.7 Övriga utvecklingmöjligheter

Som bekant genereras inte staden i realtid, men frågan är om det är möjligt att göra om systemet så att det är möjligt. Studier av tidskomplexiteten för vägnätsgenerering visar att det nog inte är helt omöjligt om genereringen sker med ett L-system som är betydligt mer optimerat än det som används i ProCity Engine. Hur som helst kräver förändringar av denna graden en stor mängd arbete och kan nog lämpa sig för ett examensarbete.

I ProCity Engine genereras staden som en enhet, men det kan tänkas att det finns intresse i att generera varje stadsdel separat. Med detta tillkommer dock vissa problem, nämligen att koppla samman dessa stadsdelar med t.ex. vägar utan att de skär sig med varandra. Det kan vara svårt för motorvägar att veta i vilken riktning den största populationen är och problem kan även uppstå när det gäller gator som stannar tvärt vid kanten på stadsdelen som kanske vetter mot ett tomt öde land, eller går rakt in i ett hus.

Interfacet till ProCity Engine och dess användarvänlighet kan verkligen diskuteras. En som inte är insatt i programmet kan lätt avskräckas från mängder av oförståeliga parametrar och textfiler som måste fyllas i. Att göra något enkelt javaprogram som tog hand om denna kommunikation är en enkel men samtidigt ur användarsynpunkt väsentlig förbättring. Detta skulle lämpligen göras med någon som kan tänkas använda programmet med inte besitter något större tekniskt kunnande, som t.ex. en 3D-designer.

4.3 Slutsats

Projektets målsättning var att visa potentialen L-system innehar när det gäller procedurell genereringen av städer, men även att försöka implementera huvuddragen i Parish och Müllers teoretiska rapport kring ämnet, samt skapa ett verktyg som kan vidareutvecklas i framtida examensarbete.

Målet var också att integrera stadsgenereringen med spelmotorn Crystal Space för att på ett tillfredställande sätt ha möjlighet att visa det som implementerats och programmerats. Anledning till att valet föll på denna var som sagt att den är tillgänglig för alla, vilket gör att den lämpar sig för t.ex. examensarbete eftersom kostnaderna då kan hållas nere. Detta gör det enklare för framtida examensarbetare att fortsätta utveckla projektet utan omständiga koverteringar till andra renderingsverktyg. Eftersom dokumentationen för spelmotorn är allt annat än komplett så krävdes efterforskningar och assistans inom vissa områden och då var Crystal Space skapare Jorrit Tyberghein väldigt hjälpsam med sin snabba respons och pedagogiska svar.

Ett datorspel som handlar om stadsbyggande, och som har dominerat sin genre sedan de släpptes 1989 är givetvis SimCity. Den uttryckskraft som finns i detta spel är något att eftersträva för ProCity Engine. Frågan är då om det är teoretiskt möjligt att inte med alltför mycket arbete skapa en stad med den mångfald och känsla som finns i SimCity 2003. Vi är definitift övertygade om att fallet är så, och inte nog med det, vi är överygade om att uttryckskraften i vårt program många gånger överstiger den hos SimCity eftersom ProCity Engine varje gång den skapar en ny stad kommer att skapa byggnader som aldrig förr skapats eller setts av någon användare, vilket gör varje resa genom en genererad stad till ett nytt äventyr. Det är möjligt att det finns någon enskild aspekt i SimCity som inte kan efterliknas, med i det stora hela så blir vi förvånade om ProCity Engine, efter möjligen någon smärre modifikation eller förbättring, inte skulle klara av de hinder som står i dess väg.

När vi konstruerade ProCity Engine försökte vi följe Parish och Müllers riktlinjer så långt det var möjligt, eller ibland önskvärt. Vissa aspekter ansåg vi alltför tidskrävade att implenetera inom tidsramen för projektet, och andra olämpliga eller onödiga.

Det viktigaste med ett stadsgenereringsverktyg är inte att varje enskild byggnad är perfekt, utan att stadskänsla är så bra att det verkligen känns som att vandra runt i en riktig stad. Speciellt överblicksbilderna på städer genererade i ProCity Engine ger verkligen en bra stadskänsla, och om man bortser från smärre texturproblem, både när det gäller byggnader och mark, så går det faktiskt att tappa bort sig i de metropoler som går att generera.

Vi ser goda möjligheter för framtida examensarbete att fortsätta där vi slutade och förbättre detta vertyg, eftersom vi är fullständigt övertygade om att potentialen för ProCity Engine är nästan obegränsad. Mycket av det som står i kapitlet om möjliga utvecklingar är relativt enkla och ger förmodligen väldigt stora visuella förbättringar, men det finns även sådant som att porta programmet till andra spelmotorer, samt utveckla nya funktioner hos programmet, så att det tar hänsyn till fler aspekter när det genererar sitt resultat.

Visar ProCity Engine att L-system har stor potential när det gäller stadsgenerering? Vi är övertygade om att så är fallet, och vi hoppas att alla som ser, testar, eller läser om programmet blir lika övertygade som oss.

Appendix

Appendix A: Användardefinierbara styrparametrar

gradient_dist_step
hw_type
hw_rule
hw_seclen
hw_rsqrsize
hw_rdkoeff
hw_plan
hw_turn
hw_bchdangle
hw_bchdangle_neg
hw_bchdist
hw_enddist
hw_bdel_low
hw_bdel_high
hw_existenz
hw_creation_ratio
hw_max_elev

st_type
st_rule
st_seclen
st_rsqrsize
st_rdkoeff
st_plan
st_turn
st_bchdangle
st_bchdangle_neg
st_bchdist
st_enddist
st_bdel_low
st_bdel_high
st_existenz
st_creation_ratio
st_max_elev

terrshow
roadshow
eyeshow

block_min_size
block_max_size
block_scan_accuracy
block_line_factor

lot_min_area
lot_biggest_factor

lot_area_disruption
lot_slice_disruption
lot_close_survival_ratio
lot_far_survival_ratio
show_lots
lots_to_terrmap
seed

create_mpp
csx
csy
csz_min
csz_max
popden_max

heightmap_name
heightmap
heightmap_mode
popmap_name
popmap
popmap_mode
nbrofit
stdens
hwdens

maxdis
best_y
best_x
mapgen_show

bd_max_level
bd_max_scale_factor
bd_min_scale_factor
bd_min_split
bd_max_split
bd_initial_scaledown_factor
bd_max_height
bd_min_height
bd_height_variation
allowed_style
max_tree_amount
tree_area
st_existenz

Appendix B: API för L-system

Som en del av examensarbeter har en generell hanterare utvecklats för utökade L-system med stöd för egendefinierade parametertyper samt externa funktioner. Exftersom programmeringsarbetet har följt en strikt modulär uppbyggnad, så kan L-system funktionaliteten utan problem användaas i andra sammanhang än just ProCity-engine. Därför följer nedan en beskrivning av vilka funktionera är och vad de gör. För att komma åt all denna funktionalitet behöver man inkludera filen: *lsystem.h*.

void lsystem_add_rule(rule_class *_rule)
Denna funktion lägger till regel till den aktuella regeluppsättningen.

char * lsystem_argname(char *name)
Lägger till ett namn bland tillåtna argumentnamn. I fall ett sådant namn redan finns, returnerar pekare till den sträng i minnet som representerar namnet. Annars returnerar en pekare till en ny sådan sträng som skapas under det aktuella anropet6.

void lsystem_clean_extarg()
Rensar de externa utargumenten, dvs argumenten från extern funktion. Detta ska alltid göras av funktionen då den vill sätta sin uppsättning av resultat som den ska skicka vidare till l-systmet.

void lsystem_clean_intarg
Rensar de externa inargumentem, dvs argument till extern funktion. Används aldrig av användaren av L-systemmodulen. Används endast av interna L-systemhanteringsfunktioner.

void lsystem_clear()
Rensar alla regler och hela L-systemsträngen. Nollställer alla variabler, så att ett nytt L-system kan initieras för körning.

void lsystem_fast()
Sätter på en extension som gör att allting går mycket fortare om man använder stora L-system. Finns dock vissa risker med detta.

argument_class * lsystem_get_extarg(int nr)
Denna funktion ska användas av den externa funktionen när den vill hämta sina argument från L-systmet. Då anger den argumentnummer och får tillbaka rätt dataobjekt. Typen på den returnerade pekaten får godtyckligt kastas om till den egenkonstruerade typ man förväntar sig att argumentet ska vara.

argument_class * lsystem_get_intarg(int nr)
Samma som ovan fast används av L-systemhanterarens inre rutiner för att hämta argument från från den externa funktionen.

module_list_class * lsystem_get_newsystem()
Returnerar det nya systemet, dvs det system som håller på att skrivas, på grudval utav det gamla.

int lsystem_get_ruc(int i)
Returnerar "rule usage count" för en specifik regel, dvs hur många gånger denna regel har används för att skriva om modul.

int lsystem_get_ruc()
Returnerar "rule usage count" för alla regler sammanlagt.

module_list_class * lsystem_get_system()
Returnerar det aktuella systemet.

char * lsystem_modname(int type)
Returnerar namet för en modultyp givet nummret på denna modultyp.

int lsystem_modtype(char *name)
Returnerar nummret på ett angivet modulnamn.

void lsystem_print_ruc()
Printar (på stdout) information om hur många gånger respektive ergel har använts.

void lsystem_print_rules()
Printar (på stdout) samtliga regler.

void lsystem_print_system()
Printar (på stdout) det aktuella systemet.

void lsystem_rewrite()
Skriver om en modul från det gamla till det nya systemet - kan men bör ej användas.

void lsystem_safe()
Stänger av den optimering som sätts på med lsystem_fast(). Därmed undviker man de eventuella problem som mdyker upp vid användningen av lsystem_fast(), programmet går dock avsevärt långsammare.

void lsystem_set_axiom(module_list_class *_axiom)
Sätter axiomet för ett givet L-system.

void lsystem_set_extarg(argument_class *arg)
Denna funktionn skall användas av den externa funktionen för att sätta argument som ska skickas till L-systemet.

void lsystem_set_intarg(argument_class *arg)
Samma som ovan fast används internt av L-systemkhanteraren för att skickar argument till den externa funktionen.

void lsystem_step()
Utför ett iterationssteg.

void _r(rule_class *)
Registrerar en regel.

void _rec(pekare till funktion)
Registrerar en extern funktion som hör till den regel som har registrerats precis innan anrop till denna funktion.

Appendix D: Rendering i Crystal Space

För rendering används en enkel Crystal Space applikation, kallad "mapviewer". Detta program följer med de exempel som inkluderas i alla distributioner av Crystal Space.

Det detta program ska göra är att läsa in en world-fil i CS-format och rendera den däri beskrivna geometrin på skärmen, samt ge användaren grundläggande möjligheter att förflytta sig runt i den tredimensionella världen.

Referenser

Litteratur

[1] Prusinkiewicy, Przemyslaw. Lindenmayer, Aristid. 1990. *The Algorithmic Beauty of Plants.* Springer-Verlag. New York.

[2] Prusinkiewicy, Przemyslaw. James, M. Mech, R. 1994. Synthetic Topiary. *SIGGRAPH 94 Conference Proceedings.*

[3] Parish, Yoav I H. Müller, Pascal. 199*. Procedural Modeling of Cities. *SIGGRAPH 01 Conference Proceedings.*

Internetsidor

[a] http://crystal.sourceforge.net/drupal/node.php?title=Features. Crystal Space funktioner. Hämtat den 25 juli 2003.

[b] http://crystal.sourceforge.net/docs/online/manual/. Crystal Space on-line dokumentation. Hämtat den 25 juli 2003.

[c] http://crystal.sourceforge.net/drupal/node.php?id=4. Crystal Space internetsida ”About Crystal Space”. Hämtat den 25 juli 2003.

[d] http://www.wikipedia.org/wiki/Game_engine. Wikipedia encyklopedi. Hämtat den 25 juli 2003.