By-generering

Interaktiv procedurel modellering

Bachelorprojekt i softwareteknologi
IMM • DTU

Simon Tobiasen
Erik Livermore

Vejledere:
Niels Jørgen Christensen
Bent Dalgaard Larsen

30. juni 2007

Abstrakt

Byer i spil er ofte statiske miljøer, der er pre-genererede af enten designere eller procedurelt-modellerende systemer. Vækst af en by er derfor simuleret som forskellige perioder, der hver har sin egen model af byen. I dette projekt er der blevet udviklet et system, der giver brugeren mulighed for at kontrollere den voksenede by. Projektet bygger på L-systemer, der har vist sig brugbare som modeller for byers udvikling.

Abstract

Cities in games are often static environments, which have been preprocessed, either by designers or by procedural engines. Growth of a city is therefore simulated as a series of eras, each with its own city model. This project has resulted in an engine that will allow for a more dynamic, procedurally growing city, letting the user influence the development. The project builds upon the research in L-systems. Algorithms built upon this, have shown to be good models for the simulation of city growth.

Lyngby
D. 30/6 - 2007

Simon Tobiasen (s042624)

Erik Livermore (s042572)

Indhold

Abstrakt 2
1 Indledning 5
1.1 Baggrund og Motivation 5
1.2 Overblik 7
1.3 Problemformulering 7
2 Forudgående arbejde 8
2.1 L-systemer 9
2.1.1 Fordel ved L-systemer 9
2.1.2 Omskrivningsalgoritmen 9
2.1.3 Tolkning 10
2.1.4 Skildpaddetolkning 11
2.1.5 L-systemer i 3 dimensioner 12
2.1.6 Stokastiske L-systemer 13
2.1.7 Kontekstsensitive L-systemer 15
2.1.8 Parametriske L-systemer 16
2.1.9 Omgivelsessensitive og åbne L-systemer 18
2.2 Pascal Müllers metode 19
2.2.1 Udvidede L-systemer 20
2.2.2 Globale mål 22
2.2.3 Lokale begrænsninger 23
3 Implementering af Beauty Of Plants 25
3.1 Systemarkitektur 25
3.2 Dynamiske regler 27
3.3 Sammenføjninger 29
3.4 Tropisme og teksturer 33
3.5 Shadere 34
3.5.1 Grene som billboards 37
4 Implementering af CityEngine 39
4.1 Systemarkitektur 39
4.1.1 L-systemet 41
4.1.2 Omgivelserne 42
4.1.3 Grafikmotor 43
4.2 L-systemet 44
4.3 Globale mål 45

4.3.1 Hovedveje 45
4.3.2 Gader 50
4.4 Lokale begrænsninger 50
4.5 Rumlig datastruktur og skæring 55
4.6 Visualisering 56
4.7 Vejnet i terræn 58
4.8 Globale variable 59
5 Egne idéer og tilføjelser 61
5.1 Mål for udvidelser 61
5.2 Udvidet brug af forsinkelse i L-systemet 62
5.3 Indsættelse af veje i L-systemet 64
6 Test og resultater 65
6.1 Datatests 65
6.2 Regeltests 67
6.3 Manipulationstest 73
6.4 Realismetest 79
7 Diskussion 80
7.1 Vurdering af resultater 80
7.2 Anvendelse af CityEngine 80
7.3 Interaktive L-systemer 81
7.4 Fremtidigt arbejde 81
7.4.1 Indsættelse af huse 81
7.4.2 Automatisk afledning af L-system 83
7.4.3 Level Of Detail 83
7.4.4 Indsættelse af træer 83
8 Konklusion 84
Appendiks 85
Kildehenvisninger 85
Figurliste 87
Introduktion til XNA 89
Kendte fejl 91

1 Indledning

Denne rapport skal dokumentere vores arbejde i forbindelse med vores bachelorprojekt i softwareteknologi omhandlende udviklingen af *CityEngine*.

CityEngine er et værktøj der kan simulere byers vækst, og lade brugeren få indflydelse på hvordan en given by udvikler sig over tid. Vores produkt adskiller sig fra andre proceduremodellerende, *content*-genererende værktøjer netop ved sin dynamik. Navnet CityEngine er således valgt fordi den ikke bare skal fungere som en *content-editor*, men snarer en motor på samme måde som f.eks. en fysikmotor.

CityEngine er implementeret i C# og XNA. Traditionelt har man ikke i særligt stort omfang benyttet *managed* sprog til udvikling af grafikapplikationer, da disse er særligt ressourcekrævende. Denne norm står måske for fald, og vores projekt har derfor også haft til formål at undersøge muligheder og begrænsninger med XNA.

Vores system baserer sig på L-systemer [1]. Vi har, for at gøre os bekendte med C#, XNA og L-systemer, implementeret programmet *Beauty of Plants*(*BoP*), der er et system til procedurel modellering af planter og især træer.

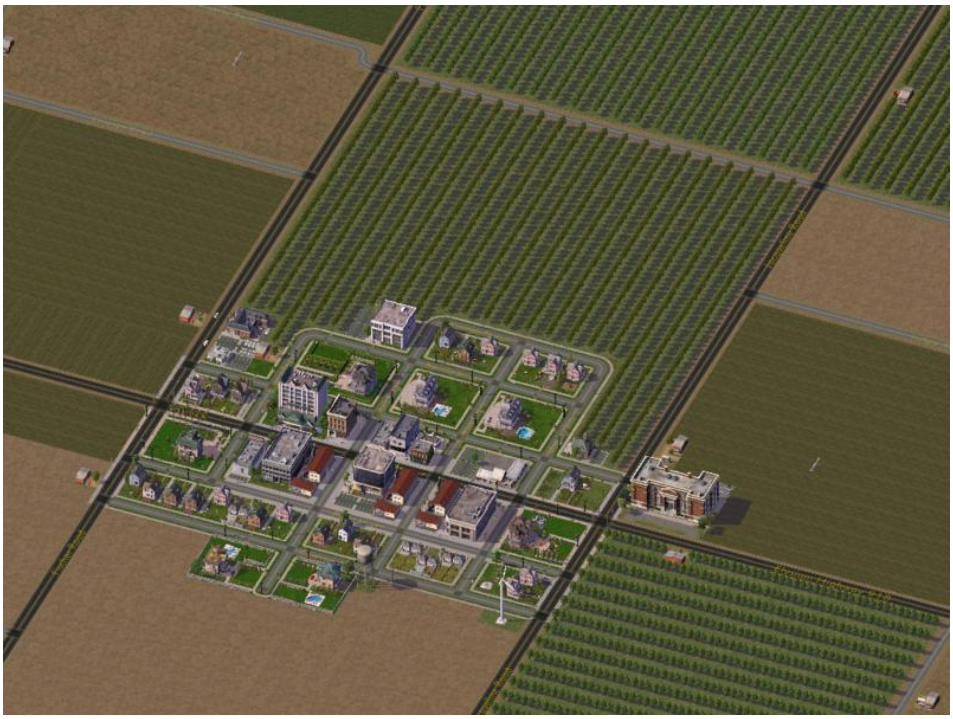
Det forventes at rapportens læsere er bekendte med pensum i kurserne 02561 *Computer Graphics* og 02563 *Virtual Reality* samt generel kendskab til softwareudvikling og computer grafik (svarende til softwareteknologi midt i studiet).

1.1 Baggrund og Motivation

Byer i computerspil er oftest statiske miljøer, der er lavet i forvejen af en grafiker, hvilket er en tidskrævende og dyr metode. En bys vækst - hvis en sådan over hovedet finder sted - vil som regel være defineret ud fra perioder, der hver er statisk definerede. Dette giver ikke en særligt ”levende” by, og giver kun tilnærmelsesvist følelsen af egentlig vækst.

Byer har igennem mange år spillet en stor rolle i computerspil. De fleste menneske befinder sig i langt størstedelen af deres liv i byer, og det er derfor naturligt at spil afspejler dette. Et klassisk eksempel er SimCity, hvoraf det første spil i serien udkom i 1989 [2]. Her har man fuld kontrol over em bys udseende, og opgaven for spilleren består i at opbygge så stor og velhavende en by som muligt. Man skal udstykke land til bebyggelse, forbinde byen med veje, bygge skole, hospitaler og parker og bestemme skattetryk. Det særlige ved SimCity er at spillet ikke kan ”vindes”, og motivationen for spilleren er simpelthen at bygge og styre hele samfundet.

I SimCity er det spillerens opgave at udbygge vejnettet og inddrage land til jordlodder, mens spillet indsætter bygninger afhængigt af hvor eftertragtet den specifikke zone er.



Figur 1 Fra SimCity 4

Der har længe eksisteret systemer i spil til automatisk generering af byer. Af de tideligste kan nævnes spil som Civilization og Transport Tycoon[3]. Især det sidstnævnte er interessant i sammenhæng med vores projekt. Spillet foregår i en verden med adskillige byer. Spilleren skal bygge et transportnetværk der forbinder disse, og derved tjene penge. Byerne vokser afhængigt af, i hvor stort et omfang deres transportbehov bliver dækket.

Figure 2 Fra Transport Tycoon

Vores ambition er at udvikle en metode, hvormed man f.eks. i spil kan udvikle byer procedurelt og samtidig påvirke disse. Systemet skal indeholde nogle af de samme elementer som de førnævnte spil implementerer, men vi vil benytte mere avancerede metoder til at gøre byerne mere realistiske.

1.2 Overblik

I afsnit 2 vil vi gennemgå hvilke metoder der hidtil er udforskede og har vist sig holdbare. I afsnit 3 og 4 vil vi gennemgå vores implementering af systemer til hhv. generering af træer og byer. I afsnit 5 vil vi redegøre for de idéer og tilføjelser vi selv har bidraget med. Vi vil, i afsnit 6, gennemgå de tests vi har udført, og samtidig vil vi redegøre for de resultater vi har opnået. I afsnit 7 diskuterer vi de mulige anvendelser vi ser for vores system, og gennemgår det arbejde vi vil udføre i den nærmeste fremtid.

Flere billeder samt videoer af projektet ligger på den medfølgende CD-rom. På denne ligger også kildekoden til programmet, og en eksekverbar version af dette.

1.3 Problemformulering

Målsætningen for vores arbejde er at udvikle en dynamisk model til beskrivelse at byvækst. By-genereringen skal være en interaktiv process hvor brugeren kan påvirke byens vækst. Projektet skal bygge på forskningen i L-systemer, og vi vil undersøge hvordan disse kan gøres interaktive. Problemformuleringen kan sammenfattes i følgende punkter:

- Byens vækst skal være hurtig, med en høj frekvens af udviklingsskridt, så det vil føles dynamisk og levende.
- Systemet skal tage højde for det landskab det udvikler sig på.
- Byens vækst skal kunne stimuleres på flere måder så brugeren har følelsen af at have kontrol over denne.
- Stimulanterne skal ikke have umiddelbar effekt, men ændre byens udseende over tid.
- Det skal være muligt at ”tegne” en by som udgangspunkt. Programmet skal så bygge videre på den. Derfor skal denne ”grund-by” kunne skrives om til et L-system.
- Vejnettet skal kunne projiceres ned på et landskab.
- Der skal udvikles en grafikmotor, der kan visualisere byen. Dette skal laves i C# og XNA
- Landskabet skal være beplantet med procedurelt genererede træer.

2 Forudgående arbejde

Modellering af virtuelle miljør, og herunder byer, til spil og film, er en meget tids- og mandskabskrævende proces. Man har defor i stor udstrækning undersøgt muligheder for, at nedbringe omkostigerne med procedurelle teknikker. Dette har også været tilfældet for modellering af bymiljøer. Som basis for projektet har vi fundet det frugtbart at studere disse metoder, med henblik på anvendelse i vores system.

Man kan, i store træk, inddele området i to hovedgrupper: En gruppe der fokuserer på genskabelse af bymiljøer fra billeddata, og en gruppe der fokuserer på generering af byer udfra forskellige parametre.

I den førstnævnte gruppe, anvendes metoder fra billedeanalyse. Mange af disse systemer har vist sig, at give virkeligt overbevisende resultater. Her kan blandt andet nævnes Takase et al. [4] og Claus Brenner [5]. Vi vil ikke anvende teknikker fra denne gruppe, da udganspunktet for vores projekt ikke skal basere sig på allerede eksisterende byer.

I den sidstnævnte gruppe, er metoder og resultater af mere varierende karakter. Dette skyldes formodentlig at målsætningen for disse systemer er mindre formaliserede, end i den førstnævnte gruppe. George Kelly og Hugh McCabe giver (i [6]) et overblik over metoder til egentlig procedurel generering af byer, der katagoriseres efter realisme, interaktivitet, effektivitet og lignende.

I [7] skitserer Greuter et al. en metode der baserer sig på simple gittermønstre, hvorimellem huse er indsat. Metoden er fuldstændigt real-time baseret, og kan således generere vilkårligt store byer. Systemet har i særdeleshed procedurel generering af huse som fokus område. Der skitseres ingen metoder til interaktion med systemet.

I 2003 har Lechner et al. [8] udviklet et system baseret på sociale agenter, der påvirker byudviklingen over tid. Veje genereres, ligesom i [7], via gittermønstre, der tilbasses de lokale omgivelser. Systemet er grundlæggende baseret på interaktivitet, og således interessant i forbindelse med vores projekt.

I [9] beskriver Mark Green et al. et system der baserer sig på deformering og sammenfletning af forskellige *template*-vejmønstre. Der skitseres ingen metoder til interaktion med sytemet over tid.

I [10] skitserer Pascal Müller en metode der baserer sig på L-systemer. I [6] vurderes det at systemet genererer særdeles realistiske vejnet. Systemet er desuden i stand til at producere meget realistiske bygninger. Der er i [10] ikke beskrevet metoder til interaktion med systemet over tid.

Valg af fremgangsmåde har været af afgørende betydning for vores arbejde. Vi har som udgangspunkt fundet metoderne i [8] og [10] attraktive.

Vi har vurderet at Pascal Müllers metode [10] producerer de mest realistiske resultater, og at vi samtidig vil være i stand til, at udvikle en interaktiv variant af denne. Vi har derfor valgt at basere vores arbejde på netop denne, og har taget udgangspunkt i hans artikel forud for SIGGRAPH i 2001 [11].

2.1 L-systemer

L-systemer spiller en helt central rolle i vores projekt, og vi vil i dette afsnit gennemgå teorien bag teknikken. Vi vil ikke komme ind på implementeringsspecifikke problemstillinger. Disse vil blive redegjort for i senere afsnit.

L-systemer har navn (L'et) efter deres idémand Aristid Lindenmayer. De har i lang tid fundet bred anvendelse ved procedurel generering af organiske modeller i computer grafik. I den følgende beskrivelse af teknikken, vil vi benytte samme notation som i *The Algorithmic Beauty of Plants* [1].

2.1.1 Fordel ved L-systemer

Hvorfor bruge L-systemer? Til at generere træer er det oplagt at bruge L-systemer. Fordelen ved L-systemet er, at de forskellige produktionsregler afspejles direkte i de forskellige komponenter af træet (stamme, grene, blade og lignende). Samtidig kan vi frakoble geometriberegninger til en senere proces. Der findes desuden god og udførlig litteratur på området, og metoden er grundigt afprøvet og er anvendt i kommercielle systemer som eksempelvis Speedtree [12].

For generering af byer er valget af L-systemer mere subtil. Vi kunne vælge at operere direkte på en grafrepræsentation af vejnettet. Dette ville formodentligt ikke være en langsommere metode, idet vi undgår operationer på strenge.

Fordelen ved at benytte L-systemer til byer er, at vi får en let og gennemsigtig notation. Vi kan med et L-system sammenfatte princippet i vores design, på få letlæselige linjer. Det er således et værktøj, der gør os i stand til let at modellere vores by på en overskuelig måde. Det er yderligere en fordel at have geometriberegninger i en separat proces, hvilket gør at man lettere kan danne sig et overblik over vores endelige program.

2.1.2 Omskrivningsalgoritmen

Et L-system består som udgangspunkt af et aksiom w, der er et sæt af *moduler* (der i vores notation repræsenteres ved tegn) og et sæt *produktionsregler*, der tilføjer moduler til aksiomet. En produktionsregel kan f.eks. være:

$$A \Rightarrow FA \qquad [1.1]$$

I [1.1] erstatter vi 'A' (*precondition*) med 'FA'. Ved evaluering af et aksiom læses hele aksiomet modul for modul. Dette foregår parallelt (i ét ”træk”), således at afledte moduler ikke er synlige for evalueringen af de resterende moduler. Hvis et modul passer en *precondition* fra en produktionsregel anvendes denne på modulet. Findes der ingen regler, der kan anvendes, bliver modulet stående, hvilket svarer til $A \Rightarrow A$. Vi opskriver normalt ikke *enhedsproduktioner* ($A \Rightarrow A$). Skal et modul udgå fra aksiomet anvendes produktionen $A \Rightarrow \varepsilon$. Anvender man f.eks. [1.1] på aksiomet AA, får vi efter første evaluering:

$$FAFA \qquad [1.2]$$

Efter anden evaluering

$$FFAFFA \qquad [1.3]$$

2.1.3 Tolkning

Et L-system kan i sit udgangspunkt minde om *context-free-grammars*. Det er først når de enkelte moduler tillægges en tolkning, at det bliver interessant i computer grafik.

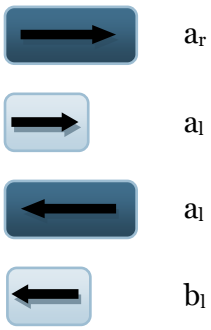
Følgende eksempel illustrerer væksten af en cellekultur kaldet *Anabaena Cantenula*. Cellekulturens livscyklus kan beskrives ved L-systemet.

w: a_r
p_1: $a_r \rightarrow a_l b_r$
p_2: $a_l \rightarrow b_l a_r$
p_3: $b_r \rightarrow a_r$
p_4: $b_l \rightarrow a_l$

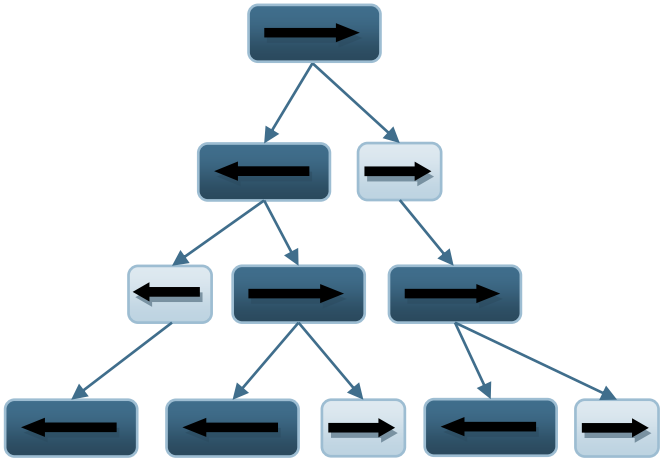
L-systemet vil generere følgende strenge:

a_r
$a_l b_r$
$b_l a_r a_r$
$a_l a_l b_r a_l b_r$
$b_l a_r b_l a_r a_r b_l a_r a_r$
...

Vi kan tolke hvert modul i strengen som en celle med bestemte egenskaber. Moduler med prædikatet a symboliserer en modercelle, der kan dele sig i to hvilket håndteres af p_1 og p_2. Der opstår således et nyt modul med prædikatet b, der symboliserer en nyfødt celle. Den nyfødte celle vokser til en modercelle ved hjælp af p_3 og p_4. Indekset l og r betegner cellens vækstretning. Vi kan visualisere cellekulturen ved at anvende følgende skema for den grafiske tolkning af hvert modul i L-systemet:



Figur 2-1
Moduler i *Anabaena Cantenula*

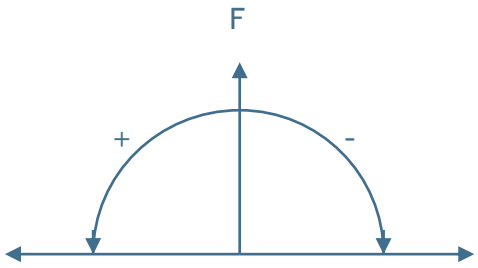


Figur 2-2 Visualisering af de 4 første afledninger af L-systemet

Når man observerer væksten af *Anabaena Cantenula* under et mikroskop, ses de cylinderformede organismer tydeligt. Cellerne deler sig skiftevist til højre og venstre som i Figur 2-2.

2.1.4 Skildpaddetolkning

En mere sofistikeret tolkning er den såkaldte *skildpaddetolkning (turtle interpretation)*. Man forstiller sig, at en skildpadde vandrer rundt på et plan. Læser vi et *'F'* i vores aksiom, går skildpadden fremad, samtidig med at den tegner en linje efter sig. Læser vi *'+'* eller *'-'* svinger skildpadden hhv. til venstre eller højre, klar til at bevæge sig frem i den nye retning. Skildpaddetolkningen er afhængig af parametre for længde (d) og vinkel (δ).

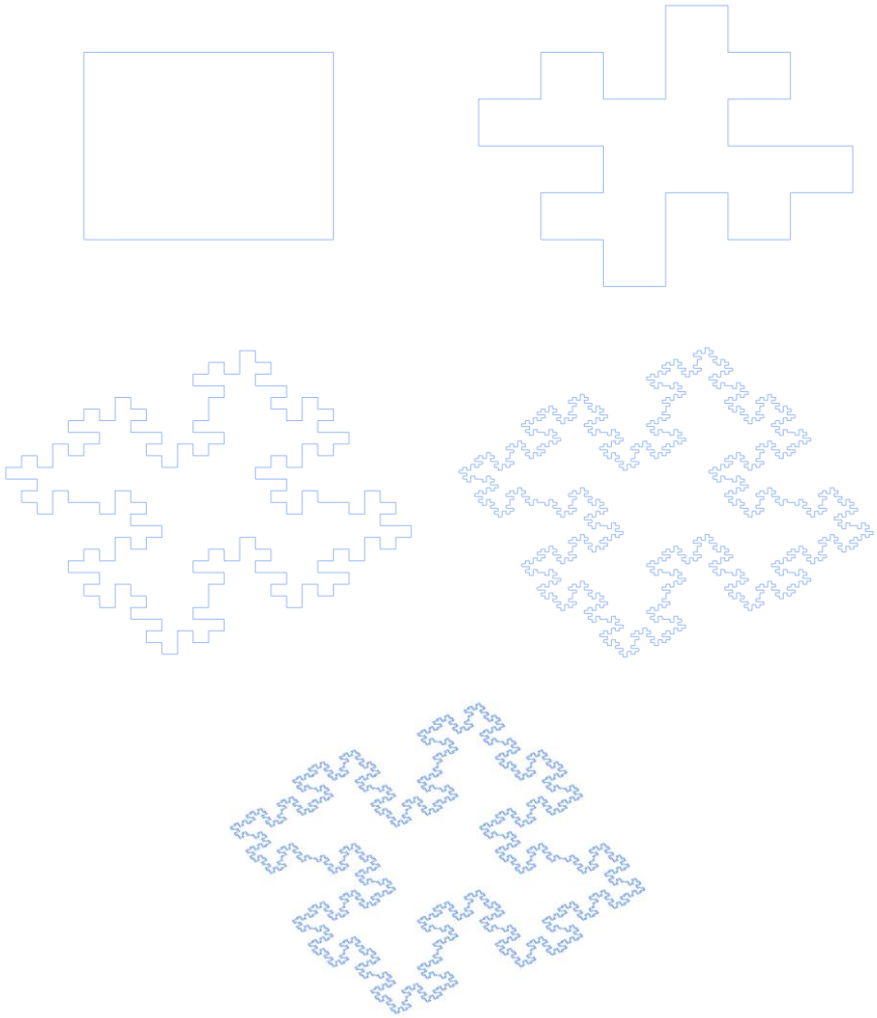


Figur 2-3 Skildpaddetolkning af modulerne '+', '-' og 'F'

Den simple skildpaddetolkning er særligt anvendelig til at beskrive fraktalkurver. Det følgende L-system beskriver det såkaldte *Koch Island* fraktal.

$\delta = 90^o$
$w:$ F-F-F-F
$p:$ F→F-F+F+FF-F-F+F

L-systemets aksiom er vist i det første trin i Figur 2-4. Skildpadden tegner en linje, hvorefter den fortager et 90° sving. Dette gentages tre gange efterfulgt af en linje, hvormed skildpadden har nået sit udgangspunkt og har tegnet et kvadrat. Læg mærke til, at figuren er sluttet i alle afledninger af L-systemet. Dette skyldes at hver kant erstattes af en linjesekvens der afsluttes i kantens oprindelige endepunkt (linjestykkerne skaleres med 1/4).



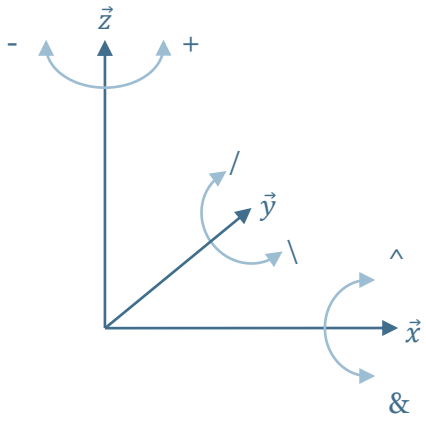
Figur 2-4 De fem første trin af *Koch Island*.
***Figurerne er genereret med Fractals*[1].**

2.1.5 L-systemer i 3 dimensioner

For at kunne modellere i 3 dimensioner, har vi brug for flere moduler som del af vores L-system. Det er klart, at vi ikke kan nøjes med rotationerne '+' og '-', da de, i den 3 dimensionelle forstand, blot foretager rotationer omkring z-aksen. Vi definerer derfor en række moduler til håndtering af hver af de tre akser (x,y,z). Figur 2-5 illustrerer dette.

+	*Roter mod venstre, omkring z-aksen*
-	*Roter mod højre, omkring z-aksen*
&	*Hæld opad, omkring x-aksen*
^	*Hæld nedad, omkring x-aksen*
\	*Kræng til venstre, omkring y-aksen*
/	*Kræng til højre, omkring y-aksen*

[1] *Fractals* er et lille program vi har udviklet for at kunne lave disse illustrationer. Det følger som kildekode, men vi vil ikke dokumentere det, da det er meget lig *BoP*.



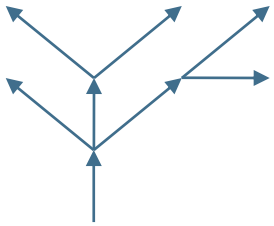
Figur 2-5 Tolkning af modulerne '+', '-', '&', '^','\' og '/'.

Indtil nu har vi set på modeller udelukkende bestående af en sammenhængende linje. I vores videre arbejde får vi brug for, at kunne modellere træstrukturer med adskillige forgreninger. I skildpaddetolkningen indesluttes en forgrening i kantede parenteser med følgende syntaks:

[	*Skildpaddens tilstand (position, retning o. lign) gemmes på en LIFO stak.*
]	*Skildpaddens tilstand erstattes med den øverste på stakken.*

Følgende aksiom med forgreninger i flere niveauer vil generere modellen i Figur 2-6,

```
F[+F][-F[-F]F]F[+F][-F]
```



Figur 2-6 Forgrenet L-system

2.1.6 Stokastiske L-systemer

Ofte har man brug for en række modeller, der alle er variationer af hinanden. Beplanter man, for eksempel en skov med nøjagtigt ens træer, oplever man et meget tydeligt artefakt, der skyldes regularitet i billedet.

Alle modeller genereret af det samme deterministiske L-system, vil være ens i deres struktur. Med stokastiske L-systemer kan vi skabe tilfældige variationer i modeller afledt fra samme aksiom. Et stokastisk L-system indeholder flere produktionsregler for samme modul. Præcist én af de mulige

produktioner vælges, hver gang vi evaluerer et modul af den pågældende type. Hvilken produktion der vælges afgøres ved lodtrækning, og hver produktion er således tilknyttet en vis sandsynlighed. Syntaksen for stokastiske produktionsregler er illustreret i det følgende L-system:

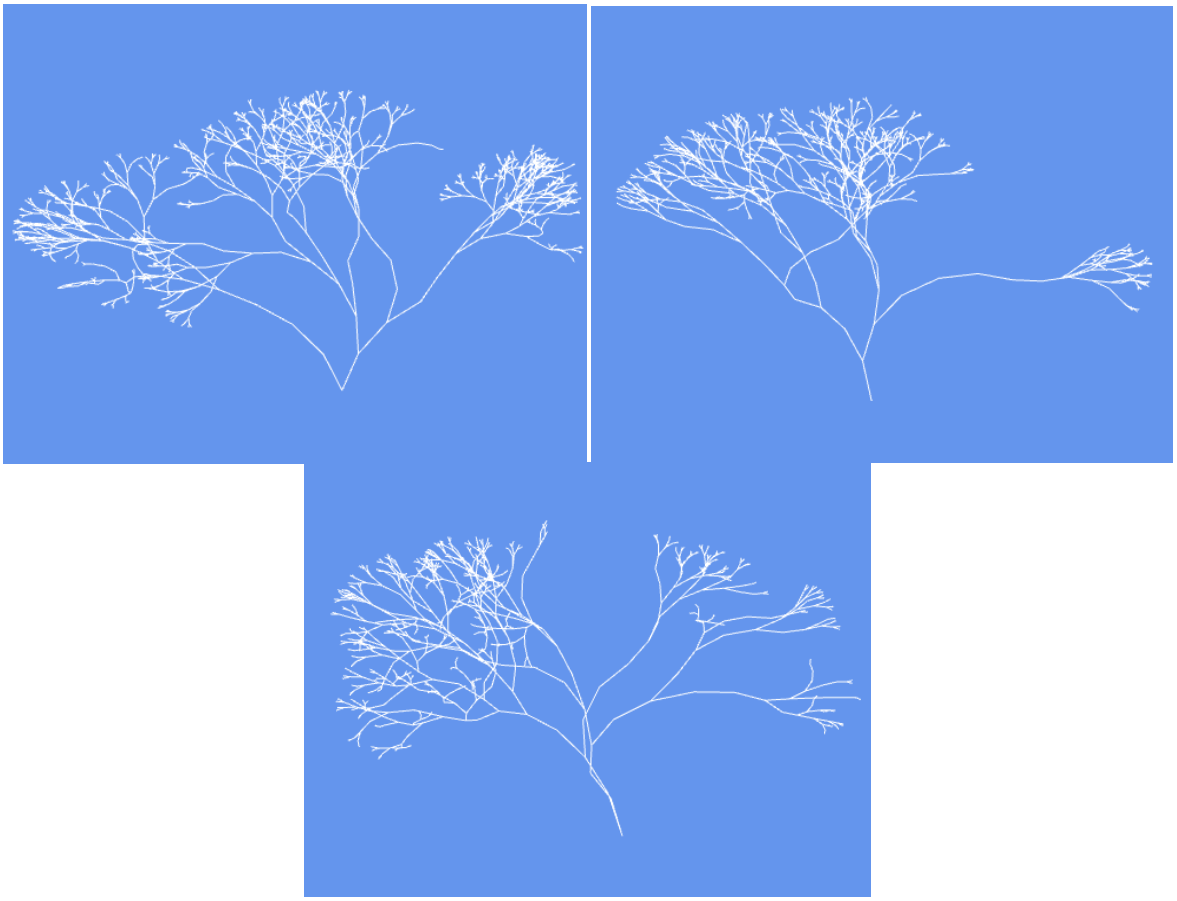
w: F
p_1: $F: 0.33 \rightarrow F[+F]F[-F]F$
p_2: $F: 0.33 \rightarrow F[+F]F$
p_3: $F: 0.34 \rightarrow F[-F]F$

En forudsætning efterfølges altså af et tal, der angiver sandsynligheden for at produktionen vælges. I dette tilfælde har vi tre muligheder, når vi møder et *F*: Enten indsætter vi en forgrening til højre, en til venstre elle begge veje.

Følgende eksempel illustrerer stokastiske L-systemer i 3 dimensioner:

$\delta=22.5^o$
w: A
p_1: $A: 0.33 \rightarrow [\&FBA]/////[\&FBA]///////[\&FBA]$
p_2: $A: 0.33 \rightarrow /////[\&FBA]///////[\&FBA]$
p_3: $A: 0.34 \rightarrow [\&FBA]$
p_4: $B \rightarrow FB$

De tre produktioner indsætter hhv. en, to eller tre forgreninger i hver sin retning. Den sidste produktion sørger for at grene, der er indsat i tidlige evalueringer, er længere end de, der er indsat i senere. Modeller genereret af dette L-system er således forskellige, afhængigt af hvilke produktioner er blevet valgt under evalueringen. Dette ses tydeligt i Figur 2-7.



Figur 2-7 Tre forskellige modeller genereret af samme L-system

2.1.7 Kontekstsensitive L-systemer

Mange plantedele gennemløber forskellige faser af vækst. F.eks. vokser et F segment ofte ikke med samme hastighed over tid. Vi kan modellere dette, ved at lade moduler i L-systemet være opmærksomme på hvilken kontekst de optræder i.

Syntaksen for kontekstsensitive L-systemer er som følger:

$$a_l < a > a_r \rightarrow X$$

Hvor a er det modul, vi læser fra aksiomet, og a_l og a_r er moduler (eller strenge af moduler) umiddelbart til venstre og højre for a. Produktionen skal tolkes som; hvis der findes et modul a med den specificerede kontekst så indsættes X i det nye aksiom.

Følgende eksempel viser hvordan et signal kan sendes gennem et aksiom:

L-systemet:

$$\begin{aligned} w: &\quad baaaaaaaaaa \\ p_1: &\quad b < a \rightarrow b \\ p_2: &\quad b \rightarrow a \end{aligned}$$

vil generere følgende strenge:

```
baaaaaaaaaa
abaaaaaaaaa
aabaaaaaaaa
aaabaaaaaaa
aaaabaaaaaa
aaaaabaaaaa
...
```

Bemærk hvordan modulet *b* bevæger sig gennem strengen.

2.1.8 Parametriske L-systemer

I eksemplet med stokastiske L-systemer så vi, hvordan en produktion kunne anvendes til at lade de enkelte segmenter vokse:

$$B \rightarrow FB$$

Vi indsætter altså altid hele segmenter af en given længde. Vi kan ikke indsætte skalerede segmenter, hvilket begrænser mængden af geometri, der kan genereres af et L-system. Vil vi f.eks. tegne en simpel trekant skal hypotenusen skaleres med $\sqrt{2}$.

For at løse dette problem associerer vi numeriske værdier til moduler. Syntaksen for parametriske moduler er som følger:

$$A(a_1,a_2,a_3,...a_n)$$

Hvor *A* er det pågældende modul med n parametre adskilt med ',’. En produktion i et parametrisk L-system kan f.eks. se således ud:

$$A(x,y) \rightarrow \&(x)F(y)\ A(x/2,y/2)$$

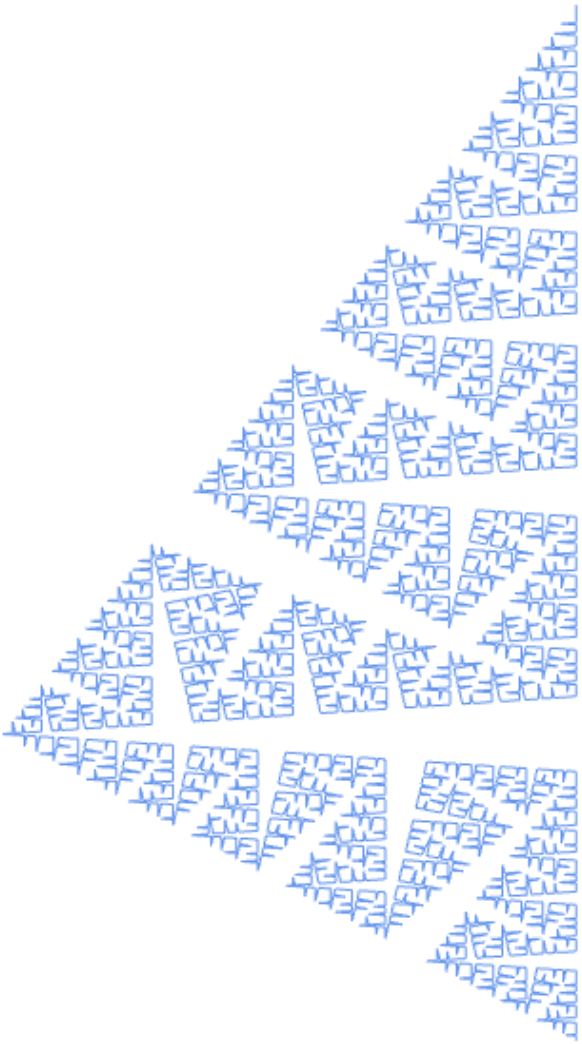
De nye moduler *&* og *F* tildeles hhv. *x* og *y* mens det ny modulet *A* får tildelt *x/2* og *y/2*. Skildpaddetolkningen for de parametriske moduler er som før, hvor værdien af parameteren angiver hvor meget der translateres eller roteres.

F(l):	*Gå fremad med længden l, hvorved en linje tegnes efter skildpadden.*
+(a):	*Roter mod venstre med vinklen a. Det samme gælder for: -(a), &(a),^(a),/(a) og \(a) (omkring de tilknyttede akser)*
[,]:	*Har ingen tilknyttede parametre i skildpaddetolkningen.*

I parametriske L-systemer kan der optræde logiske udtryk på parametrene, der gør de enkelte produktioner betingede sammen med stokastiske og kontekstsensitive betingelser. Dette illustreres ved følgende L-system:

$\delta = 85^o$
$c=1;\ p=0.3;\ q=c-p;\ h=\sqrt{q*p}$
$w:\quad F(1,0)$
$p_1:\quad F(x,t) : t==0 \rightarrow F(x*p,2)+F(x*h,1)--F(x*h,1)+F(x*q,0)$
$p_2:\quad F(x,t) : t>0 \rightarrow F(x,t-1)$

Variablene *c*, *p*, *q* og *h* definerer en trekant med grundlinje *c*, højde *h* og afstand *p* fra hjørne til *h*-vinkelret på grundlinjen. L-systemet genererer modellen i Figur 2-8.



Figur 2-8 Parametrisk L-system (model genereret i Fractals)

Betingelserne i p_1 og p_2 sikrer, at de korte segmenter forsinkes i deres omskrivning. Parameteren t tæller således ned til nul hvorefter p_1 træder i kraft. Denne metode er vigtig for vores projekt, og vil blive diskuteret senere.

2.1.9 Omgivelsessensitive og åbne L-systemer

Enhver plante er påvirket af det miljø, den vokser i. F.eks. har planter behov for lys, og grenene vil derfor forsøge at vokse sådan, at bladene får så meget lys som omgivelserne tillader det. Planten påvirker samtidigt sine omgivelser. F.eks. skygger plantens blade for andre planter, og dens rødder suger vand fra jorden, hvilket giver mindre gunstige forhold for naboplanter. Disse fænomener kan modelleres ved omgivelsessensitive og åbne L-systemer. I det følgende vil vi behandle disse under et.

Til at kommunikere med omgivelserne indsættes et kommunikationsmodul.

Kommunikationsmodulet har syntaksen:

$$?A(a_1,a_2,a_3,\ldots a_n)$$

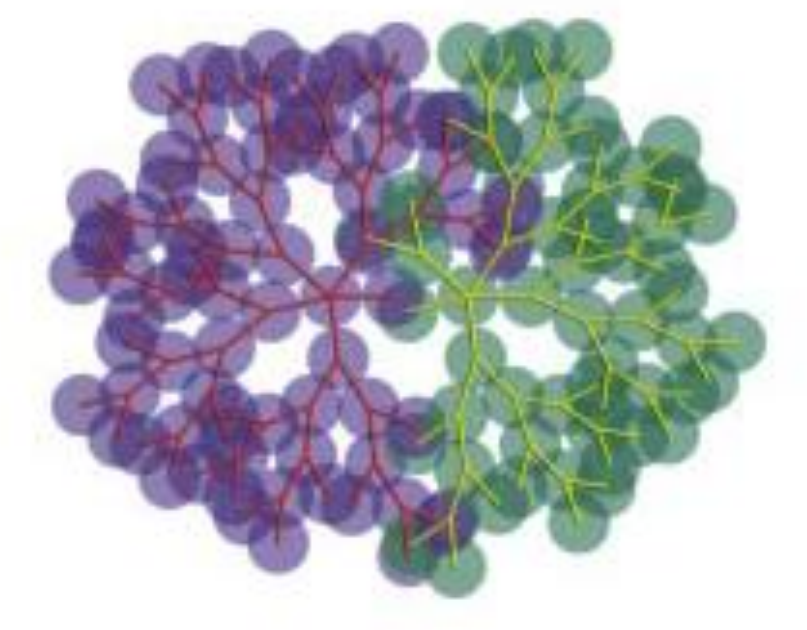
Alle kommunikationsmoduler markeres med et $?$, og parametrene a_j bruges til at sende og modtage beskeder fra omgivelserne. Alt afhængigt af modulets type (A = P, H, R, U eller lignende) sendes skildpaddens position eller orientering til omgivelserne. Hvilke af disse værdier der sendes med, specificeres i en beskrivelse af kommunikationsmodulet.

En vigtig forskel fra almindelige moduler og kommunikationsmoduler er, at parametre kan være uinitialiserede umiddelbart efter indsættelse i L-systemet. De manglende variable indsættes ved en efterfølgende gennemgang af L-systemet, der ikke har indflydelse på aksiomets form. Systemet tolkes i denne gennemgang således, at position og retning for skildpadden kendes, når vi møder et kommunikationsmodul. Følgende eksempel illustrer brugen af åbne L-systemer:

To grene vokser nær hinanden, og for at opnå de mest gunstige lysforhold, vil de, så vidt det er muligt, vokse i hver sin retning. Omgivelserne modelleres således, at hver spids indsættes i det todimensionelle plan, og spidsen tildeles en radius omkring dette punkt. Kommunikationen mellem L-system og omgivelser foregår gennem kommunikationsmodulet: $?E(x)$, hvor skildpaddens position sendes, sammen med parameteren x, der er udtryk for hvor "energisk" planten vokser. Ud fra disse oplysninger bestemmer miljøet spidsens videre vækst: Returneres $x=1$ vokser spidsen videre i to forgreninger (p_1). Returneres $x=0$ ophører spidsen med at vokse (p_2). Til denne beslutning tages der højde for om spidsen falder inden for en eksisterende spids' radius, og for forskellen mellem hvor energiske de to spidser er. L-systemet er angivet i den følgende boks, og billedet under, er hvordan situationen tager sig ud. System og billede fra [13].

```
δ= 138.5°
 r1 = 0.94; r2 = 0.87; a1 = 24.4; a2 = 36.9
w:      -(90)[F(1)?E(1)A(1)]+(δ)[F(1)/?E(1)A(1)]
        +(δ)[F(1)?E(1)A(1)]+(δ)[F(1)/?E(1)A(1)]
        +(δ)[F(1)?E(1)A(1)]
p1:     ?E(x) < A(v) : x == 1→
        [+(a2)F(v*r2)?E(r2)A(v*r2)] -(a1)F(v*r1)/?E(r1)A(v*r1)
p2:     ?E(x) → ε
```

L-systemet genererer følgende model, hvor radius for de indsatte spidser er medtaget:



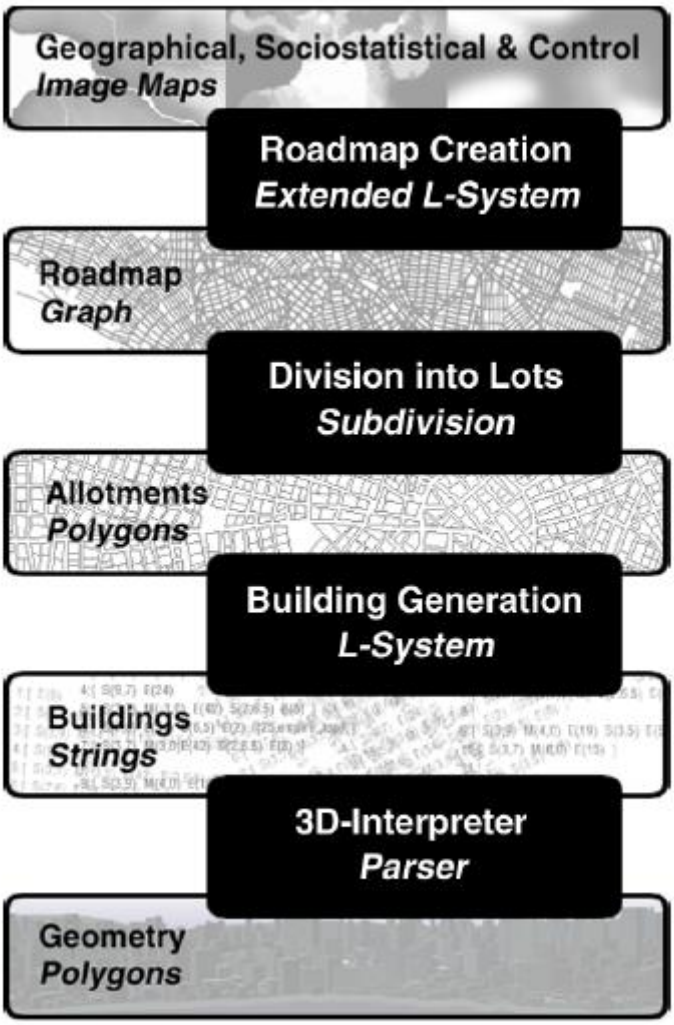
Figur 2-9 To grene konkurrerer om lys

2.2 Pascal Müllers metode

Tyskeren Pascal Müller har udviklet et system til generering af byer [14], der i vid udstrækning baserer sig på L-systemer. Han har lavet en del arbejde i forbindelse med sin uddannelse, hvor han har skrevet to større opgaver om emnet ([10] og [15]). Han har fremvist sit arbejde på SIGGRAPH i 2001 [11] og 2006 [16], og vil fremvise sin nyeste forskning på samme i 2007. Det er hans artikel fra '01 [11] vi har taget udgangspunkt i.

Systemet kan udfra en relativt lille mængde kortdata generere en hel by, bestående af et vejnet samt procedurelt genererede bygninger. Han bruger et højdekort over landskabet for at undgå, at byer kommer til at ligge op ad bjerge og lignende. Et kort, der fortæller hvor der er søer og hav, gør det muligt at få kystveje og broer, samt undgå at veje går ud i vandet. Til sidst er der et kort, der illustrerer en ønsket befolkningsfordeling. Disse kombineres, og hans program laver et realistisk vejnet baseret på det givne data. Efter dette er gjort, bliver vejnettet delt ind i jordlodder ud fra de

fundne veje og vejkryds. Disse jordlodder bruger han til at sætte huse ind i landskabet, og dermed skabe meget realistiske byer. Figur 2-10 illustrerer systemets opbygning hovedtræk.



Figur 2-10 Oversigt over Müllers system

En vigtig del af Müllers arbejde har været, at undersøge og formalisere virkelige byers struktur. Først og fremmest skelner han mellem hovedveje og gader. Hovedveje forbinder større befolkningsområder, mens gader forbinder beboede områder med hovedvejene. Gaderne følger ofte et dominerende gademønster og danner blokke. Hovedvejene kan følge forskellige overordnede mønstre, som vi vil komme nærmere ind på.

Vores projekt har ikke til formål at undersøge virkelige byer, men vi baserer vores arbejde på hans ideer. Vores arbejde omfatter desuden udelukkende generering af vejnet. Vi har valgt at bygge vores system på Müllers metode, da vi mener at han har opnået yderst realistiske resultater.

2.2.1 Udvidede L-systemer

Vi har taget udgangspunkt i den første del af artiklen, omhandlende udvikling af et realistisk vejnet udfra givne parametre. Pascal Müller har udviklet et L-system, som han bruger til at beskrive byvækst som vejnet (se nedenfor), og vi bruger dette som grundlag for vores projekt.

Han har videreudviklet L-system-konceptet til at kunne tage højde for mange parametre, mens antallet af produktionsregler holdes lavt. Dette opnås ved at lade reglerne kalde funktioner. Disse funktioner sætter parametre i modulerne, og influerer dermed L-systemet. Funktionerne kan returnere værdier, der indikerer at et modul skal fjernes. En produktion sørger for, at dette bliver gjort i den følgende iteration.

Funktionen, *globalGoals*, sætter værdier for vejens længde og retning. Funktionen bruger forskellige data, der angives af brugeren, og vægter disse for at opnå den ønskede overordnede vejstruktur. De globale mål kan indikere at vejen skal fjernes, hvis der ikke kan findes en anvendelig placering. Dette kan være tilfældet, hvis der ikke kan findes en rimelig hældning, og vejen derfor kommer til at gå op ad en meget stejl bjergvæg.

Når vejen er rettet ind efter de globale mål, betragtes lokale begrænsninger. L-systemet kalder en funktion, der ser på om vejen rent faktisk kan ligge, hvor den skal ifølge de globale mål. De lokale begrænsninger undersøger først og fremmest om vejen passer ind i det allerede genererede vejnet. Der ses på, om vejen krydser andre veje, eller om den burde rettes ind, så den rammer et vejkryds. De lokale begrænsninger kan også forkaste vejen, hvis der ikke kan findes en lovlig position til vejstykket. Det kan for eksempel være, at den går direkte ud i havet.

L-systemet

w: R(del, initialRuleAttr)?I(initRoadAttr, UNASSIGNED)

p_1: R(del, ruleAttr) : del<0 → ε

p_2: R(del, ruleAttr) > ?I(roadAttr, state) : state == SUCCEED
{globalGoals(ruleAttr, roadAttr)}
→ +(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[1],UNASSIGNED)

p_3: R(del,ruleAttr)>?I(roadAttr,state) : state == FAILED → ε

p_4: B(del,ruleAttr,road) : del>0 → B(del-1,ruleAttr,roadAttr)

p_5: B(del,ruleAttr,roadAttr) : del==0→ [R(del,ruleAttr)
?I(roadAttr,UNASSIGNED)]

p_6: B(del,ruleAttr,roadAttr) : del<0 → ε

p_7: R(del,ruleAttr) < ?I(roadAttr,state) : del<0 → ε

p_8: ?I(roadAttr,state) : state==UNASSIGNED
{localConstraints(roadAttr)}
→ ?I(roadAttr,state)

p_9: ?I(roadAttr,state) : state!=UNASSIGNED → ε

Pascal Müllers L-system til vejnet

L-systemet er bygget op af 9 produktionsregler, og følger notationen fra [1]. Aksiomet initialiserer L-systemet med et vejmodul(*R*) og et indsættelsesmodul(*?I*). Disse moduler optræder altid sammen i L-systemet, og håndterer et vejsegment og det komunikationsmodul der styrer, om det skal indsættes i vejnettet. Reglerne p_1, p_2 og p_3 styrer vejmodulet. p_2 kontrollerer forgrening, ved at indsætte to forgreningsmoduler(*B*), et vejmodul og et indsættelsesmodul. De globale mål initialiserer de parametre, der bliver brugt som udgangspunkt for vejstykkerne, og gemmer dem i lister. Listerne *pRuleAttr* og *pRoadAttr* indeholder data, som bliver brugt i de nye moduler (*B*, *R*, og *?I*). Forsinkelse (*del*, for "delay") bliver brugt til at bremse udviklingen af L-systemet, og de lokale begrænsninger signalerer til L-systemet, at en vej skal fjernes ved at sætte vejens *state* til *FAILED*, og p_3 eksekverer dette.

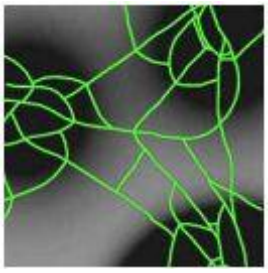
Reglerne p_4, p_5 og p_6 styrer forgreningsmodulet, tæller *del* ned og indsætter et vejmodul når den når 0. Dermed kan udviklingen styres ved at ændre startværdien for *del*. p_4 tæller *del*-variablen ned i det tilfælde, at den er højere end 0. p_5 instantierer vejmodulet sammen med det tilhørende indsættelsesmodul når *del* har nået 0. Vejens *state* sættes til *UNASSIGNED* sådan, at vi i det næste skridt kan rette vejen ind efter de lokaler begrænsninger.

Reglerne p_1 og p_7 sletter et vejmodul hvis *del* er negativ. De globale mål returnerer et negativt tal, og dermed fjernes vejen fra systemet. Regel p_8 og p_9 sætter de endelige værdier for vejmodulet ifølge de lokale begrænsninger, og sørger for at den færdige vej bliver indsat i vejnettet.

2.2.2 Globale mål

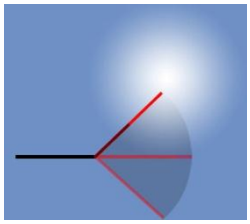
De globale mål bruges til at sætte modulernes udgangsparametre. Der bruges forskellige kort og regler, der vægtes individuelt, for at skabe den ønskede effekt.

Et intensitetskort over befolkningstæthed gør det muligt, at have flere bycentre på et landskab. Vejene er mere tilbøjelige til at vokse hen mod steder, hvor der er høj befolkningstæthed, og hen mod bycentre. Dette gør, at der vil være mange veje samlet omkring bycentrene og i områder med høj befolkningstæthed, og færre veje der så forbinder disse centre (se Figur 2-11). Dette er meget virkelighedstro, da der som regel kun er få veje imellem byer, men mange i og omkring disse byer.



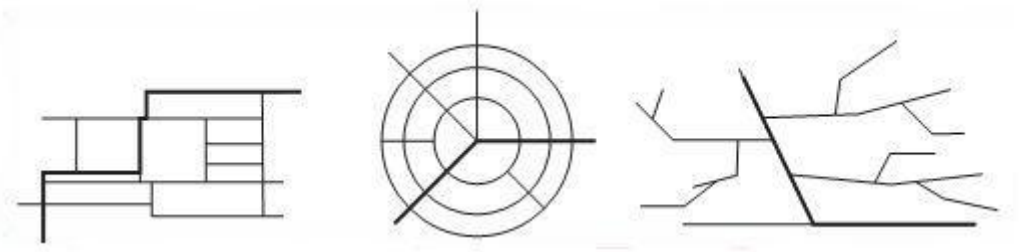
Figur 2-11 Billede fra [11]

Denne effekt opnås ved at forsøge med forskellige retninger i en vifte omkring den først foreslåede (se Figur 2-12). Hvis der er en retning, der resulterer i en højere befolkningstæthed, bliver denne foretrukket.



Figur 2-12 Den mulighed med højst intensitet bliver valgt

Herefter evalueres regler for hvordan vejnettet skal se ud. Det vil sige, hvilken overordnet struktur vejene skal følge. For eksempel har København ringveje omkring indre by, samt radiale veje der fører trafik til og fra centrum. Manhattan i New York er derimod opbygget efter et gitter, og følger denne struktur slavisk. Müller har opstillet fire regler, som hans byer så kan følge: ”*Basic rule*” hvor vejene kun tager højde for befolkningstætheden, og ikke følger noget overordnet mønster. ”*New York rule*” hvor vejene strengt følger førnævnte gitter, og har en helt bestemt længde. ”*Paris rule*” minder om København, hvor der er et bycentrum, og så radiale veje ud fra dette centrum, og koncentriske veje der danner ringe rundt om. Det sidste ses oftest i gamle byer, hvor centrum af byen ligger, som det har gjort i mange hundrede år, og store veje først er kommet til senere. Til sidst er der ”*San Francisco rule*”, hvor vejene følger et højdekort i stedet for et mønster. Vejene vil så vælge de steder, hvor stigningen er mindst og slet ikke kunne eksistere, hvis de når over en hvis hældning. Så længe landskabet er helt plant, vil denne ligne ”*basic rule*”, men hvis der er bjerge og/eller bakker vil reglen skabe byer, der går uden om disse.



Figur 2-13 Tre eksempler på regler, der giver meget forskellige vejnet. Billede fra [11]

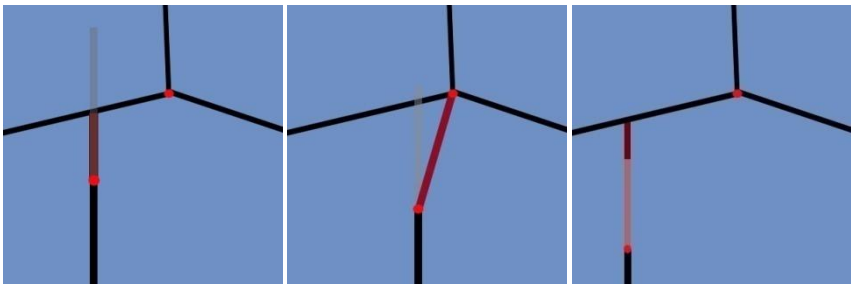
Befolkningstæthedskortet, og de forskellige regler kombineres og vægtes som man ønsker. For eksempel kan en del af en by følge et bestemt mønster, og en anden del være påvirket af en anden regel, og derfor se meget anderledes ud. Dette giver en effekten af, at byen er vokset over lang tid, hvor forskellige byplaner og andre faktorer har spillet ind.

2.2.3 Lokale begrænsninger

Efter at de potentielle veje har fået rettet deres retning og længde ind efter de globale mål, bliver de undersøgt for lokale begrænsninger. Disse lokale begrænsninger finjusterer så vejene, så de kommer til at ligge helt som de skal. Ændringer kan ske, hvis en vej ender i et område, hvor den ikke må vokse. Det kunne være en park eller ud i havet. De lokale begrænsninger bevirker, at vejen bliver rettet til i længde eller retning så dette ikke sker, eller slettet hvis der ikke kan findes en løsning.

I en virkelig by er blinde veje undtagelsen frem for reglen, men med det foreslåede L-system vil disse opstå hyppigt. Der er derfor taget højde for dette ved at gøre L-systemet ”selvsensitivt”. Dette gør at veje vil finde andre veje samt vejkryds og så rette sig ind efter dem (se Figur 2-14). De lokale begrænsninger sørger altså for, at forkorte veje der krydser hinanden, samt forlænge veje der

næsten når hen til en anden vej. Desuden undersøges der for omkringliggende vejkryds, og vinklen ændres så vejen støder op til et eventuelt nærliggende kryds. Dette skaber effekten af vejløkker, hvilket giver meget realistiske kareer, og gør at der kun sjældent opstår blinde veje.



Figur 2-14 Hhv. vejkrydsning, vejkryds undersøgelse, og forlængelse

Denne metode til bygenerering har vist sig meget god, og anvendes for eksempel Introversion [17], der arbejder på en *City Generator* (beskrevet i et forum indlæg i [18]). Den er hurtig og relativt let at implementere, og giver et meget realistisk resultat. Der er dog også problemer med metoden. På grund af den uundgåelige tilfældighed vil der altid opstå tomme områder uden nogen veje i. Dette kan ligne parker eller andre større områder uden mindre veje, men kan også blive urealistisk hvis de opstår for hyppigt. Desuden kan det komme til at se lidt mærkeligt ud i den yderste kant af byen. Her kan der opstå små grupper af gader der ”stikker ud” af byen. Dette ses tydeligt på Figur 2-15, som er et udklip af et billede fra Müllers artikel [11].

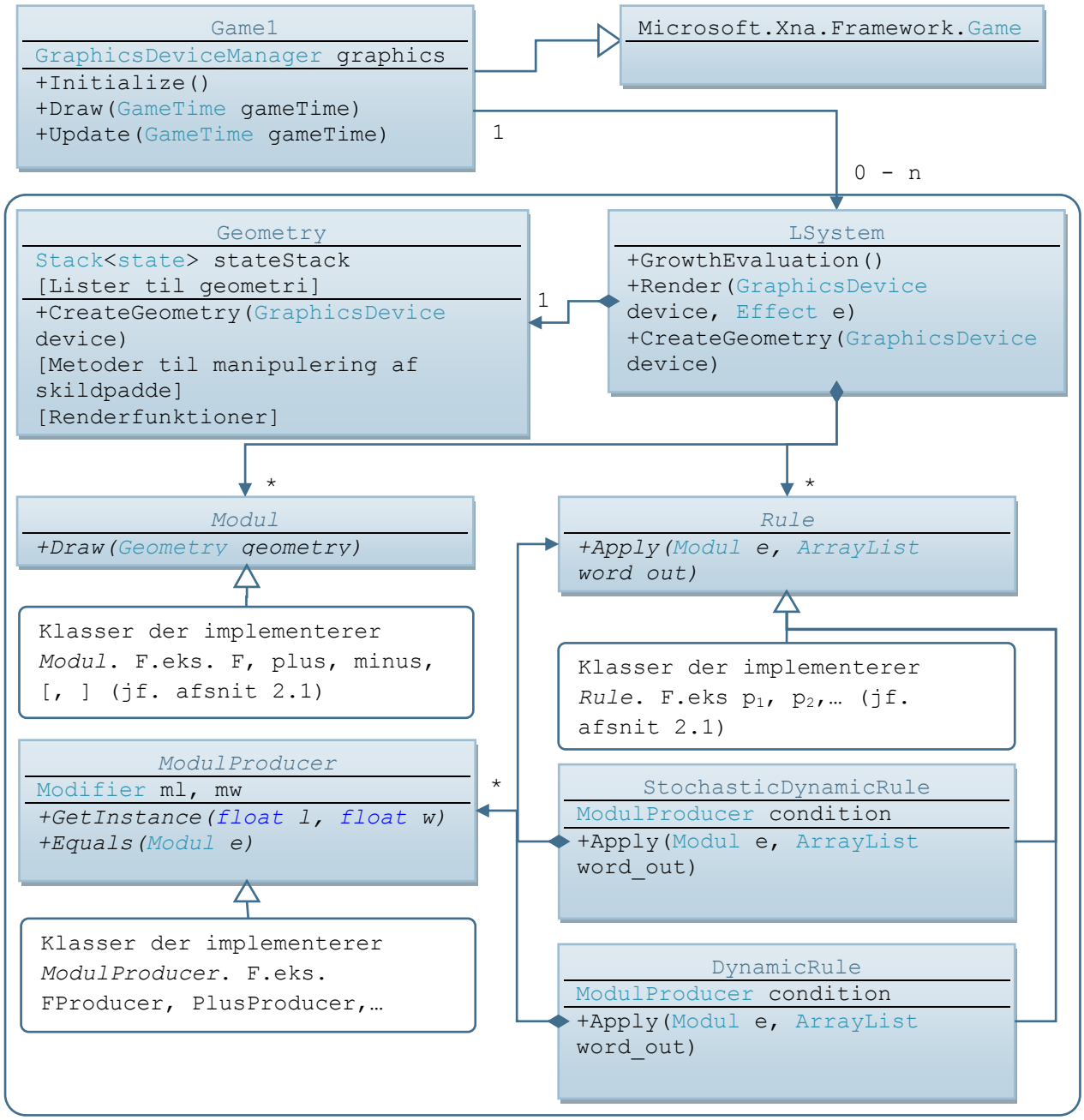
Figur 2-15 Fejl i kanten af byen

3 Implementering af Beauty Of Plants

Beauty of Plants (*BoP*) er et system til generering af planter, og bygger derfor direkte på den teori vi gennemgik i afsnit 2.1. Systemet er implementeret i C# med grafikbiblioteket XNA. Et af hovedformålene med at udvikle *BoP* var, at gøre os selv bekendte med L-systemer og XNA.

Det overordnede krav til vores implementering er, at vi skal kunne generere vellignende træer, der skal indgå som inventar i de byer vi genererer. Vi vil derfor udover det selvstændige program også inkorporere L-systemet og den visuelle fortolker i CityEngine.

3.1 Systemarkitektur



Figur 3-1 UML-diagram over BoP

C# er et objektorienteret programmeringssprog, og for at give et overblik over programstrukturen vil vi her præsentere et forsimplet UML-diagram sammen med en kort beskrivelse af de enkelte klassers opgaver. Vi har udeladt hjælpeklasser og hjælpefunktioner for at programstrukturens hovedtræk bedre kan illustreres.

Game1 nedarver fra et XNA `game`-objekt. Alle XNA programmer har et `game`-objekt, hvor bl.a. metoderne `Initialize`, `Draw` og `Update` findes. *Game1* instantieres i *Program*-klassen hvor `main`-metoden findes, hvorefter game-løkken startes i sin egen programtråd. `Draw`-metoden er en callbackfunktion der kaldes af XNA-frameworket, med en frekvens der alene bestemmes af XNA. Vi overrider `Draw`-metoden, og kalder vores egne renderingsmetoder herfra. `Initialize` og `Update` overrides på samme måde og bruges til hhv. at initialisere alle ikke-grafiske ressourcer (f.eks. L-systemet) og fortage opdateringer af programmets tilstand (f.eks. igennem en GUI). *Game1* indeholder desuden vores L-system(er).

LSystem er klassen, der repræsenterer hele L-systemet. L-systemet indeholder aksiomet, der er en liste af moduler og et regelsæt, der er en liste af regler. Disse udgør tilsammen det specifikke L-system. Den centrale algoritme for L-systemet, er implementeret i metoden `GrowthEvaluation`, der simpelthen itererer over alle moduler i aksiomet, og gennemgår alle regler i ordnet rækkefølge ved at kalde `Apply` med det specifikke modul som parameter. For at reglerne opfatter evalueringen, som foregående i ét træk, benytter vi to lister, sådan at vi skriver nye moduler til en liste, der var tom ved start af evalueringen. Efterfølgende kopierer vi denne liste til aksiomet. Hvis ingen regler kan anvendes på det pågældende modul, tilføjer vi det til slut, hvorved vi har taget højde for enhedsproduktionen. Vi tolker L-systemet i metoden `CreateGeometry`, ved simpelthen at iterere over alle moduler og kalde deres `Draw`-metoder.

LSystem indeholder desuden et *Geometry*-objekt, der på én gang repræsenterer skildpadden og samtidig skal foranstalte manipulering og rendering af den genererede geometri. Vi kunne have valgt at lade geometriklassen være kendt af *Game1*-klassen, hvormed L-systemet ville generere et geometriobjekt, som *Game1*-objektet kunne kalde `render`-metoden på (på den måde ville geometriklassen være en slags meshklasse, der ville kunne indgå i en grafikmotor). Vi har foretrukket at lade *LSystem* håndtere interfacet til *Game1*-klassen, da det på den måde bliver mere overskueligt når træerne skal inkorporeres i *CityEngine*. Desuden ville den første løsning ikke være særligt intuitiv, når man tænker på træer som levende organismer, der kan vokse, hvormed man kan betragte L-system og geometri som en entitet.

Geometry indeholder metoder til rendering, der kaldes af L-systemet, og metoder til generering af geometrien. Den genererede geometri opbevares i lister (f.eks. `trunkVertices`) af passende vertexformater, indtil de sendes til en vertexbuffer når metoden `CreateGeometry` kaldes af L-systemet.

Skildpaddens tilstand repræsenteres i geometriklassen ved datastrukturen *State*, som består af en matrix, der repræsenterer skildpaddens position og orientering, samt en breddeparameter der repræsenterer segmenters tykkelse. Disse smides løbende på og hives af en stak af states efterhånden som '[' og ']' optræder i L-systemet. Når et modul kalder `AddSegment`, tilføjes et nyt segment til geometrien med hensyntagen til den nuværende skildpaddetilstand. Dette gøres ved at transformere alle punkter fra segmentets lokale koordinatsystem over i orienteringsmatricens koordinatsystem.

Modul er den abstrakte klasse, der repræsenterer et modul i L-systemet. De specifikke moduler nedarver fra denne klasse og overrider den abstrakte metode `Draw`. I `Draw` implementerer hvert modul sin egen tolkning. F.eks. kalder modulet 'F' `AddSegment` i geometriklassen, og modulet '+' ganger en rotationsmatrix på den nuværende matrix.

Rule er den abstrakte klasse, der repræsenterer en regel i L-systemet. På samme måde som for moduler nedarver en specifik regel fra denne klasse. På denne måde har vi L-systemer, der kun kan specificeres i design-time. De enkelte regler modtager et modul sammen med en output-liste. Reglen kontrollerer om modulet er af den rette type (opfylder produktionens betingelse), og i så fald indsættes de specifikke moduler i enden af outputlisten.

3.2 Dynamiske regler

For at kunne specificere L-systemer i runtime har vi implementeret dynamiske regler. Dette gør os i stand til at hente et L-system fra f.eks. en fil.

DynamicRule nedarver fra `Rule`, og implementerer `Apply`. I stedet for at hardcode hvilke moduler, reglen skal producere, har vi en liste indeholdende objekter af typen *ModulProducer*. Vi implementerer en *ModulProducer* for hver type modul. En producer bruges til at instantiere moduler af den pågældende type, ved et kald til `GetInstance`-metoden. I den dynamiske regel kan vi nu løbe alle producers igennem, og på den måde instantiere objekter som kan indsættes i outputlisten. For at kunne modificere parametre (f.eks. skalere eller addere dem) har vi imlementeret en række *Modifiers* (+,-,*, konstant samt en speciel tilfældigheds modifier der specificerer en maksimum og minimums værdi). Vi har valgt en simpel begrænsning, der betyder, at vi maksimalt kan have to parametre pr. modul og at 1. parameter i det indkommende modul, kun kan anvendes som 1. parameter i de tilføjede moduler. Det samme gælder for 2. parameter. Det betyder, at følgende produktion f.eks. ikke er tilladt:

$$A(x,y) \rightarrow B(y*0.3,\ x*1.1)$$

Mens produktionen

$$A(x,y) \rightarrow B(x*0.3,\ y*1.1)$$

er tilladt. Dette er sjældent et egentligt problem, når vi modellerer med L-systemer, og det gør implementeringen betydeligt simplere; vi lader simpelthen den angivende modifier modificere parameteren, inden vi sender værdien til det nye modul.

StochasticDynamicRule gør os i stand til også at have dynamiske regler, der er stokastiske. Den stokastiske regel indeholder en liste af andre regler, der trækkes lod om ud fra en vægtning, der er associeret med regler i datastrukturen *SubRule*.

Vi har implementeret en parser, der kan tage et L-system skrevet i XML-format, og returnere et instantieret *LSystem*-objekt. Vi kan f.eks. specificere følgende L-system

rot = 137,5; wScale = 0,75; l = 50; lScale = 0,9

w: A(1,1)
*p_1: A(l,w)→!(rot)!(w)F(l)A(lScale*l,wScale*w)*

ved XML dokumentet:

```
<lsystem name="Tree">
  <defines>
    <define name="rot" val="137,5" />
    <define name="wScale" val="0,75" />
    <define name="l" val="50" />
    <define name="lScale" val="0,9" />
  </defines>

  <axiom>
    <modul type="A" param1 ="1" param2 ="1" />
  </axiom>

  <rule name="p1" condition="A">
    <out type="/">
      <modifier type="constant" param="rot"/>
    </out>
    <out type="!">
      <modifier type="passOn/>
    </out>
    <out type="F">
      <modifier type="passOn/>
    </out>
    <out type="A">
      <modifier type="mult" param="lScale" />
      <modifier type="mult" param="wScale" />
    </out>
  </rule>
```

Her starter vi med at definere de variable, vi har brug for i et seperat `rod`-element. De parsede variable gemmes i en hashtabel, så vi kan slå dem op direkte, når vi støder på dem i den efterfølgende parsing af L-systemet.

Aksiomet har sit eget `rod`-element `<axiom>`, hvori vi lister aksiomets elementer i hvert sit XML-underelement.

En regel indledes med `rod`-elementet `<rule>`, der har *tags*, der betegner navnet og hvilken betingelse, der skal opfyldes, før reglen træder i kraft. Navnet bruges, når vi refererer til reglen i f.eks. en under-regel i et stokastisk L-system (reglerne gemmes i en hashtabel). Efterfølgende defineres outputelementer sammen med deres modifiers.

Bemærk at f.eks. modulet '!' udelukkende tager 2. parameteren, mens 'F' udelukkende tager 1. parameteren. `passOn` fortæller, at det pågældende parameter ikke skal modificeres. En liste over semantikken for de forskellige moduler og modifiers kan ses i appendiks.

En stokastisk regel har syntaksen:

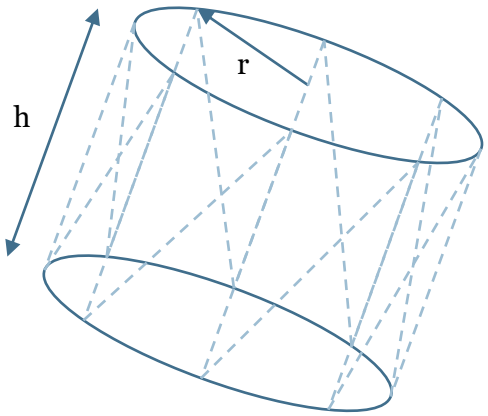
```
<storule name="s1" condition="B">
   <subrule name="p1" weight="0,4"/>
   <subrule name="p2" weight="0,6"/>
 </storule>
```

Hvor de enkelte regler listes sammen med deres vægt.

Parseren er implementeret i klassen *XMLParser*.

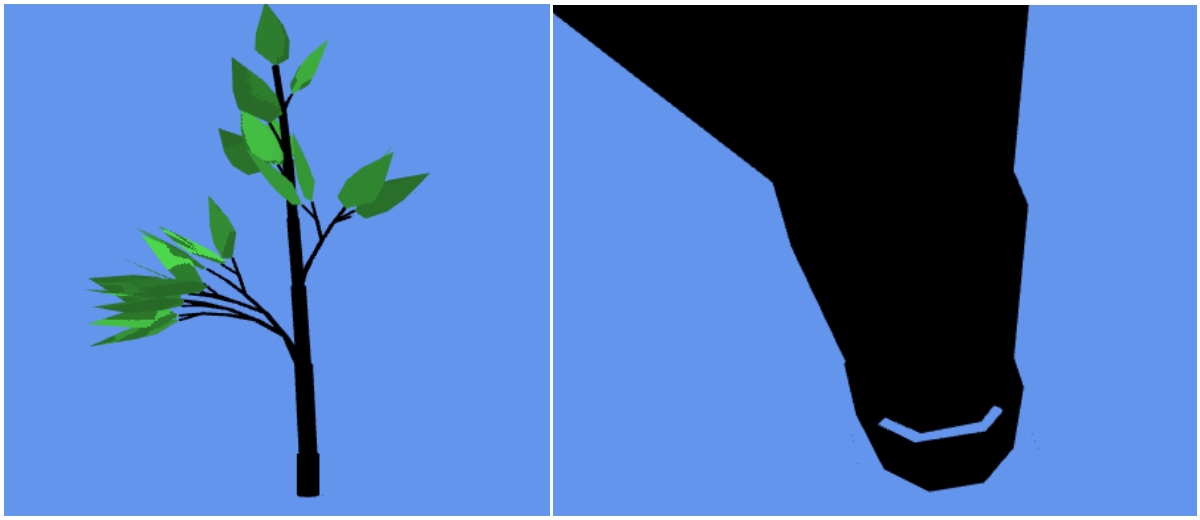
3.3 Sammenføjninger

Et vigtigt element i at forbedre det visuelle indtryk, er at lave pæne sammenføjninger mellem indsatte segmenter. Som udgangspunkt renderer vi segmenter som en cylinder ved et triangle-strip, som vist i Figur 3-2.



Figur 3-2 Tegning af et segment

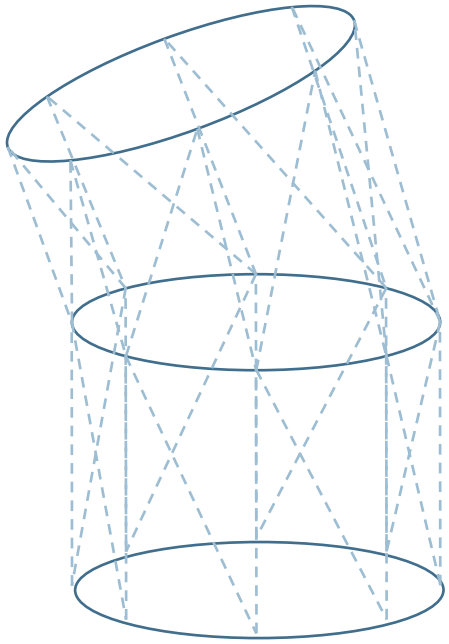
Dette giver umiddelbart et udmærket visuelt indtryk, og metoden tager højde for længde og bredde af et segment. Resultatet kan ses i Figur 3-3.



Figur 3-3 Segmenter renderet som cylindere.

På afstand er sammenføjningerne ikke synlige (Figur 3-3 til venstre), men når man nærmer sig, bliver artefakterne voldsomt forstyrrende (Figur 3-3 til højre).

Vores idé til sammenføjningerne baserer sig på, at rendere kegler i stedet for cylindere. For at undgå fuger mellem segmenter referer vi tilbage til vertices i det tidligere segment, sådan at polygonerne "fortsætter" over i det nye segment, som det er illustreret i Figur 3-4.

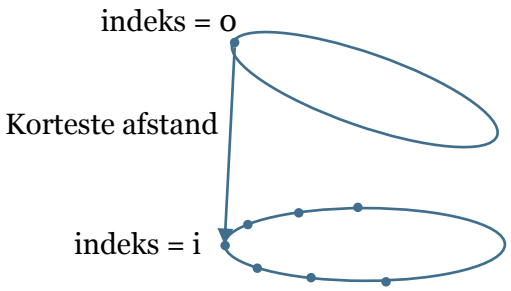


Figur 3-4 Segmenter føjet sammen ved at dele vertices

I vores implementering gemmer vi simpelthen en pointer til den sidst renderede vertex i vores skildpadde-state. Metoden giver i mange tilfælde et udmærket resultat, men ofte foregår der rotationer omkring stammen hvilket forårsager synlige artefakter, der tydeligt ses i Figur 3-5.

Figur 3-5 Rotations artefakt

Vores løsning på dette problem er simpelthen at undersøge det tidligere segment for at finde den vertex, der ligger nærmest den vertex, i det nye segment, der har indeks 0 (Figur 3-6). Disse to punkter vil i langt de fleste praktiske tilfælde være det mulige sæt, hvorom rotationen er mindst.



Figur 3-6 Indeksering mellem to segmenter

Vi ved nu hvor stor en rotation, vi skal korrigere for i vores indeksering i det tidligere segment. En simpel funktion kan afgøre hvilket indeks i det foregående segment, der svarer til et givet indeks i det nye:

```
addedIndex = (i + indexOfSmallest) % (detail)
```

Her er `i` det pågældende indeks vi vil finde en nabo til. `detail` er antallet af vertices, der omkranser et segment og `indexOfSmallest` er det indeks vi fandt, da vi søgte efter den korteste afstand.

Til trods for, at metoden er yderst simpel, giver den nogle forbavsende gode resultater. I figur **Error! Reference source not found.** ses sammenføjningen mellem grenene.

Figur 3-7 Sammenføjning mellem grene

Det kræver nogle meget akavede rotationer for at metoden fejler, og den nærmeste nabo ikke svarer til den korrekte indeksering. Problemet med metoden er, at ved kraftige rotationer omkring segmentets normal og binormal vil mellemsegmentet blive mast fladt, hvilket er illustreret i Figur 3-8.

Figur 3-8 Fladt segment forårsaget af rotation

Dette er dog sjældent synligt, da rotationen omkring disse akser, for det meste, er relativt lille.

3.4 Tropisme og teksturer

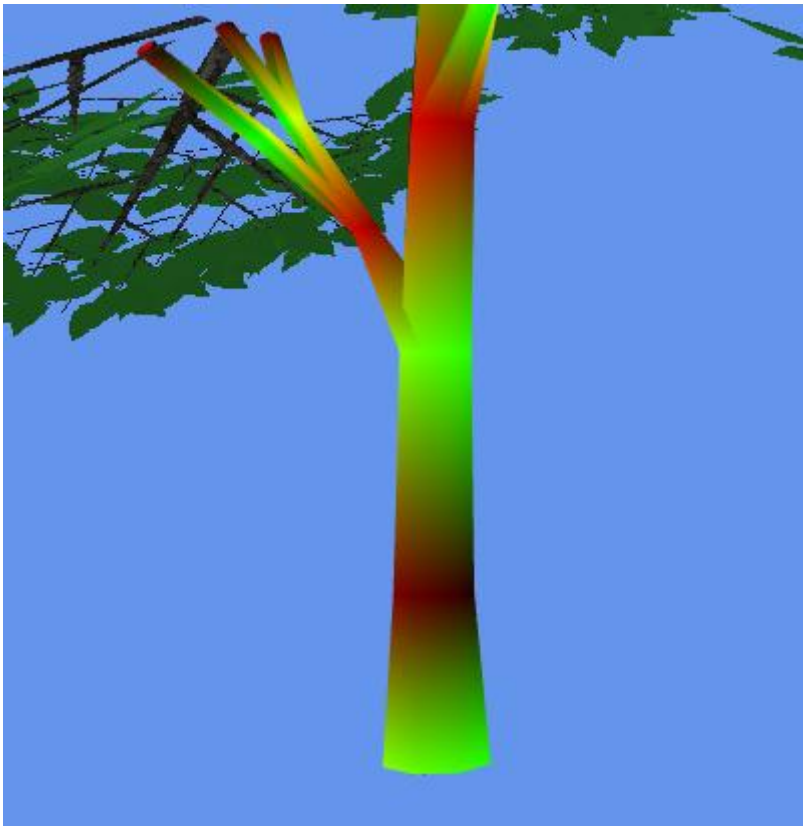
To simple mekanismer, der er med til at forbedre det visuelle indtryk markant, er tropisme og teksturer.

Tropisme er en bøjning af grenene i en bestemt retning. Det kan f.eks. være tyngdekraften eller vind som forårsager dette. I vores implementering repræsenteres tropisme ved en retningsvektor T og en skalar værdi α. Vi roterer så hvert segment i samme retning som T, med en vinkel der er proportionel med α og længden af segmentet.

Vi finder aksen hvorom vi skal rotere fra krydsproduktet mellem T og skildpaddens retningsvektor. Dermed har vi en akse i verdenskoordinater, der efterfølgende transformeres ind i skildpaddens normalrum. Vi kan nu beregne en rotationsmatrix og gange denne på skildpaddens orienteringsmatrix. Vi ser eksempler på anvendt tropisme:

Figur 3-9 Tropisme. Venstre: Gravitation. Højre: Vind.

I Figur 3-9 ses det, at vi med teksturer kan give indtryk af stammer med bark. Vi har valgt at gentage teksturen på hvert enkelt segment, men spejle den sådan, at vi undgår en grim søm. Vi bruger vores skildpadde-state til at gemme teksturens t–koordinat ved slutningen af det foregående, som vi således kan fortsætte (spejlet) i det nye (Figur 3-10).



Figur 3-10 Teksturkoordinater; s = rød, t = grøn

3.5 Shadere

I XNA findes ingen *fixed function pipeline* (bortset fra en *basic effekt*). Vi har derfor implementeret shadere til alle dele af vores træer i HLSL(*lsyseffect.fx*).

Hvert blad renderes som en flade. Til hver flade er der tilknyttet en enkelt fladenormal, hvilket gør at lyset er ens på begge sider af bladet. Som vist i Figur 3-11 giver dette ikke det korrekte indtryk af hvordan lyset falder på kronen.

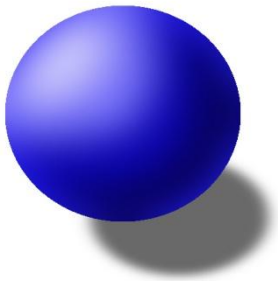
Figur 3-11 Sammenligning. Lyset kommer fra samme side på begge billeder

Vi kan undersøge fra hvilken side vi betragter bladet, ved at kigge på vinklen mellem øjevektoren og normalen. Er vinklen større end 90°, inverterer vi normalen. Da blade ofte er meget tynde, vil en stor del af lyset sive igennem. Derfor fikserer vi ikke lysbidraget mellem 0 og 1, i stedet benytter vi følgende formel for den diffuse koefficient

```
max(ldotn,abs(ldotn)*0.4)
```

der bevarer 40 % af det diffuse lys på bagsiden af bladet. Dermed har vi en mere korrekt lysberegning på begge sider af fladen.

En anden vigtig effekt er kronens skygge på sig selv. En meget tæt krone vil have en belysning der minder om en kugle: Lys på den side der vender mod lyset, og mørkere på den anden.



Figur 3-12 Belysning af kugle

For træets krone kan denne effekt, kun opnås ved skygge, da den består af mange enkelte blade. Vi benytter den almindelige shadow map algoritme [19]. Vi har valgt denne teknik, fordi den ikke for alvor bliver langsommere når mængden af geometri bliver høj (i modsætning til shadow volume algoritmer [20]).

Figur 3-13 Trækrone der kaster skygge på sig selv

Ved at lade en del af lyset skinne igennem, der hvor der ikke er skygge, opnår vi desuden en god effekt, når træet ses i silhuet. For træet i Figur 3-11 vil kronen se ens ud, uanset hvorfra man betragter den. Mens med den nye metode, får vi et resultat som i Figur 3-14. Omkring kanten skinner mere lys igennem end i de områder hvor kronen er bred.

Figur 3-14 Trækrone i silhuet

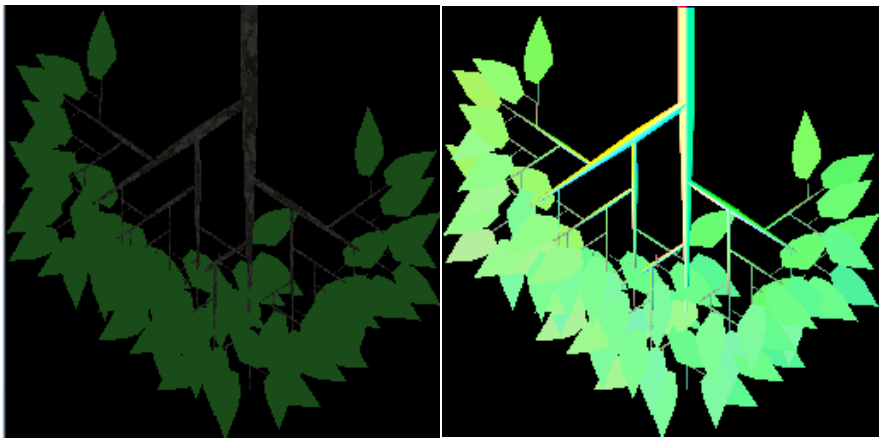
3.5.1 Grene som billboards

For at gøre træerne mindre geometri-tunge (de kan let bestå af mere end 1.000.000 vertices) har vi indført et modul i vores L-system, der, når det tolkes, tegner et billboard:

> *T(x,y): Tegn et billboard i skildpaddens nuværende position og orientering med bredde = x og højde = y*

Vi har gjort det muligt for brugeren at specificere et L-system, der skal renderes til det pågældende billboard i XML-filen. Vi tegner udelukkende det pågældende L-systems diffuse farver i billboardet, mens normalerne tegnes til et normal map. Når billboardet tegnes på træet, beregnes dets TBN-matrix (**T**angent, **B**itangent og **N**ormal), og vi kan nu fortage den samme lysberegning (i TBN-koordinatsystemet), som vi gjorde for bladene.

Billboardteknikken er velkendt og benyttet af bl.a. *SpeedTree* [12]. Det er svært at måle effekten, fordi billboards giver et anderledes vissuelt indtryk. Vores tests peger på, at vi kan regulere antallet af vertices i en model fra 400.000 ned til omkring 20.000, og stadig beholde samme "kompleksitet" i billedet. Vi vil desuden argumentere form at vi med billboards har muliggjort syntesen af langt mere realistiske træer.



Figur 3-15 Diffusemap og normalmap

Det endelige resultat ser således ud:

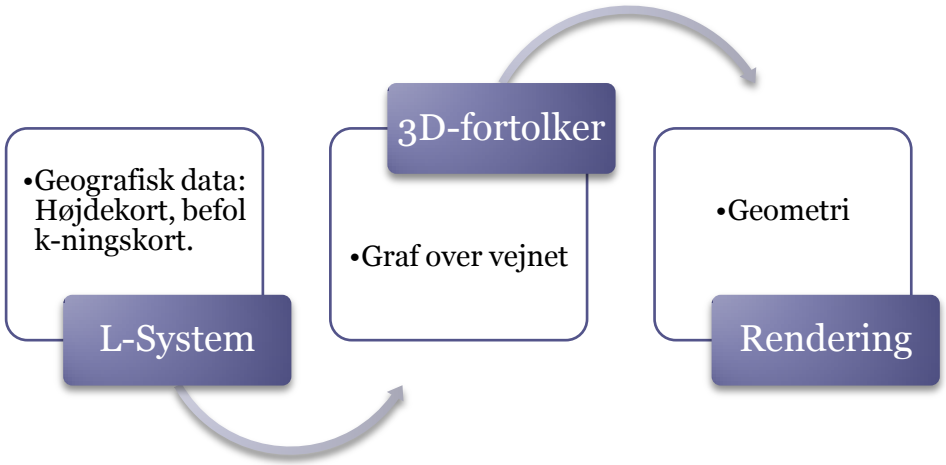
Figur 3-16 Det endelige resultat

4 Implementering af CityEngine

4.1 Systemarkitektur

Vi har, til CityEngine, brugt de erfaringer vi gjorde med implementeringen af *BoP*, hvilket afspejles i strukturen af programmet. Programmet er, ligesom *BoP*, udviklet i C# og XNA. Vores strategi har været først at implementere Müllers metode, hvorefter vi kunne bygge oven på denne med egne ideer.

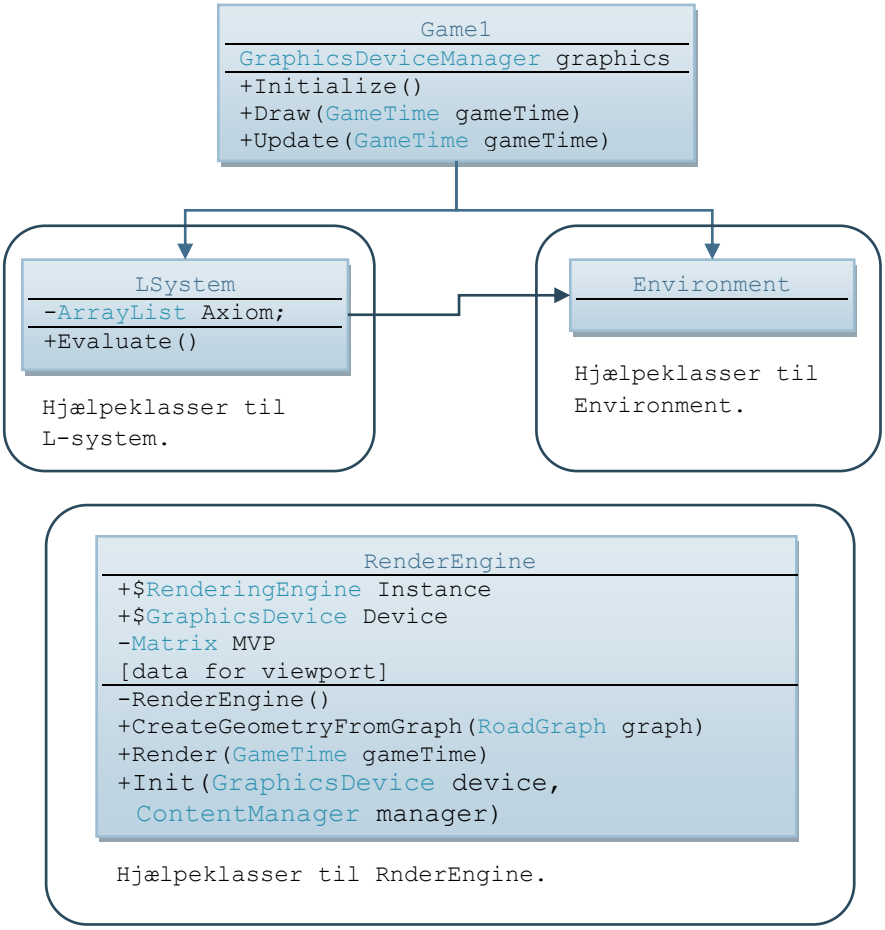
For at give et overblik over systemet, vil vi her præsentere et forsimplet UML-diagram. I Vores program viderefører vi strukturen, som den er præsenteret i Müllers artikel [11]. Da vi har valgt ikke at indsætte huse, ser vores pipeline for systemet ud som følger:



Figur 4-1 Illustraion af pipeline'en i vores system

Da vores system er interaktivt gentages denne proces flere gange, men med langt færre iterationer i L-systemet. Vi renderer altså mellemprodukterne af byen, fra aksiomet til en færdig by (dvs. færdig i Müllers system).

I vores system er de tre delprocesser repræsenteret ved klasserne: *Lsystem*, *Environment* og *RenderEngine*. Sammenhængen mellem de tre dele, kan beskrives ved følgende UML-diagram (som i UML-diagrammet for *BoP* er hjælpe-klasser og -funktioner udeladt):



Figur 4-2 UML-diagram for de tre delprocesser

Som i *BoP* har vi en XNA `Game`-klasse, der håndterer game-løkken. Ved opstart sørger `Game`-klassen for instantiering af et *LSystem* og et *Environment*.

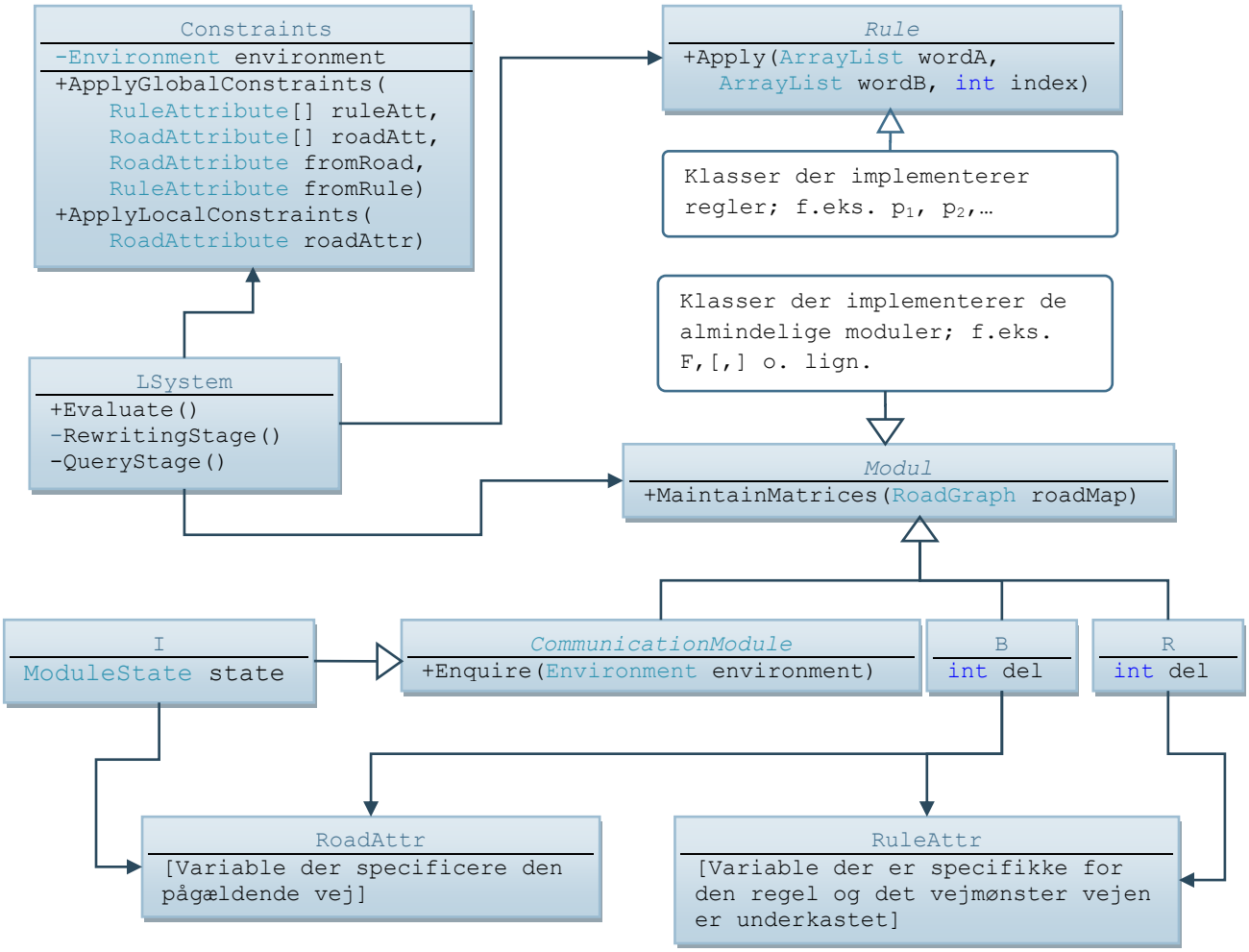
LSystem er et L-system i den almindelige forstand.

Environment er komposition af flere moduler, der udgør de omgivelser som L-systemet befinder sig i. *Environment* indeholder både det geografiske data og L-systemets fortolker, sammen med den graf der er produktet af L-systemet.

RenderEngine er en grafikmotor, der har til opgave at rendere den graf der produceres i L-systemet. I vores design har vi valgt at modellere *RenderEngine* som en `singleton`, hvilket vil sige at klassen indeholder en statisk readonly-pointer til en instans af sig selv. Denne instans oprettes ved kald af `Init`, hvor den private constructor kaldes. Vi har valgt dette design, fordi det sikrer at vi kun har én instans af *RenderEngine* i hele systemet. Alle hjælpeklasser har desuden adgang til selve grafikmotoren, uden at vi skal sende den med som parameter. De fleste klasser har brug for resourcer eller metoder i *RenderEngine*; f.eks. har vandeffekten brug for at rendere en reflektionstekstur, og derfor undgår vi at skulle sende en stor mængde parametre rundt ved metodekald.

Vi vil i det følgende kort gennemgå arkitekturen for de tre dele af systemet.

4.1.1 L-systemet



Figur 4-3 UML-diagram over arkitekturen af L-systemet

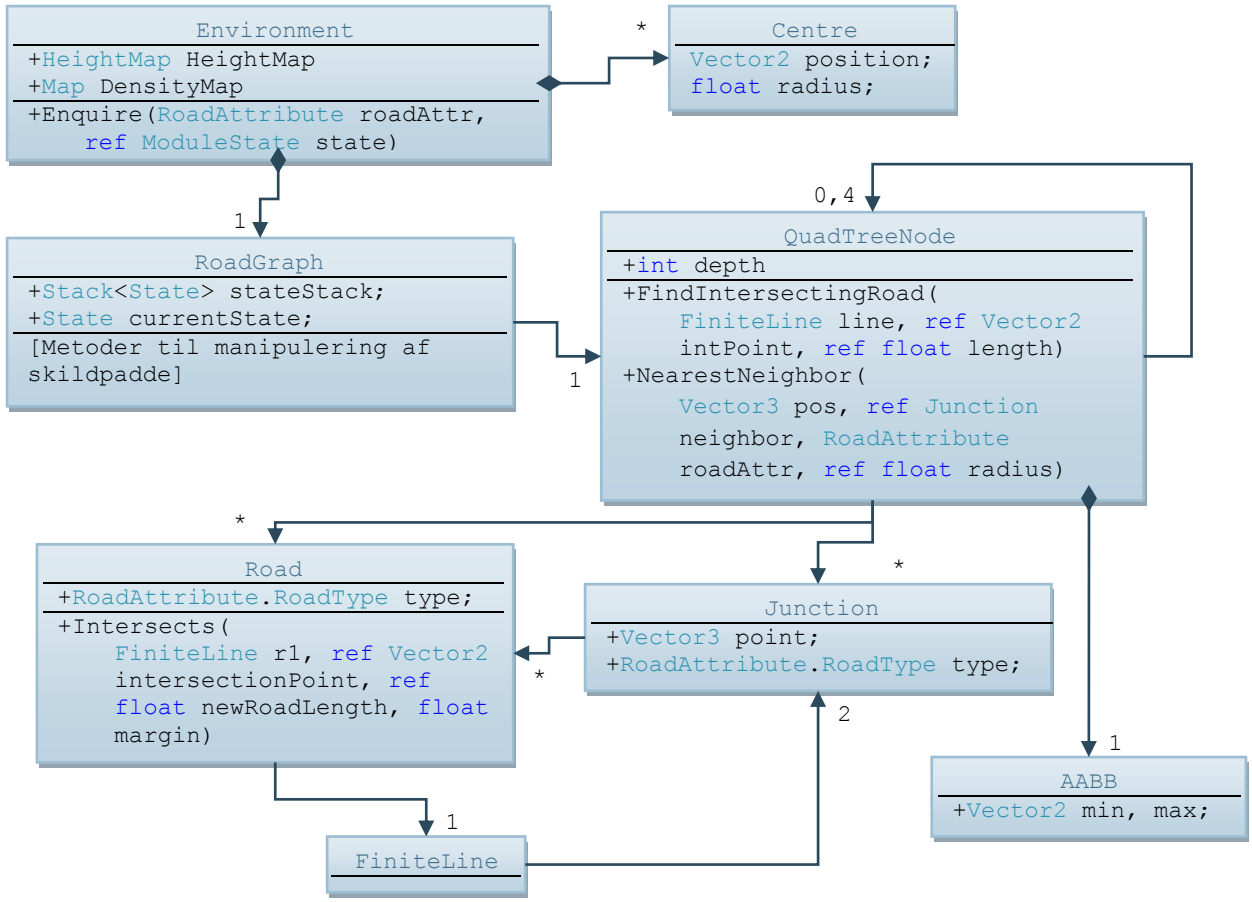
Som det fremgår, minder L-systemet i *CityEngine* meget om det vi implementerede i *BoP*. Vi har de abstrakte klasser *Rule* og *Module*, som specifikke moduler og regler nedarver fra. Systemet baserer sig på ét statisk L-system, der kan modelleres via inputdata, og der er således ikke behov for dynamiske regler.

Til forskel fra *BoP* er dette L-system et såkaldt åbent L-system, og vi har derfor brug for et kommunikationsmodul, der her er repræsenteret ved den abstrakte klasse *CommunicationModule*. Denne klasse har den abstrakte metode `Enquire`, der håndterer kommunikationen med omgivelserne. Det åbne L-system kræver en evaluering af L-systemet i to faser: `RewritingStage` og `QueryStage`, der kaldes af *LSystem*. Da vi finder den matrix, der definerer en forgrening under Query-fasen (og gemmer den i `RoadAttr`), er der ikke behov for en egentlig fortolknings fase, idet vi kan skrive til den endelige vejnets-graf, det øjeblik en vej har fundet sin endelige position. Derfor har *Module* ikke en `Draw`-metode, men en `MaintainMatrices`-metode. I den endelige udgave af vores L-system har vi ligesom i Müllers system [11] kun ét kommunikationsmodul kaldet *I*.

Udover de almindelige moduler der blev gennemgået i afsnit 2.1, har vi modulerne *R* og *B* der, som i Müllers artikel [11], repræsenterer hhv. en vej og en forgrening i L-systemet.

Klassen *Constraints* varetager justeringen af nye veje i forhold til hvilke begrænsninger der findes i omgivelserne. Bemærk at metoderne `ApplyGlobalConstraints` og `ApplyLocalConstraints` befinder sig i hver sin fil, der har partielle klasser af *Constraints*.

4.1.2 Omgivelserne



Figur 4-4 UML-diagram over omgivelserne

Som nævnt repræsenterer *Environment* de omgivelser L-systemet befinder sig i. Klassen indeholder et højdekort, og et kort der repræsenterer befolkningstætheden. Vi har desuden en liste over de centre der er placeret af brugeren.

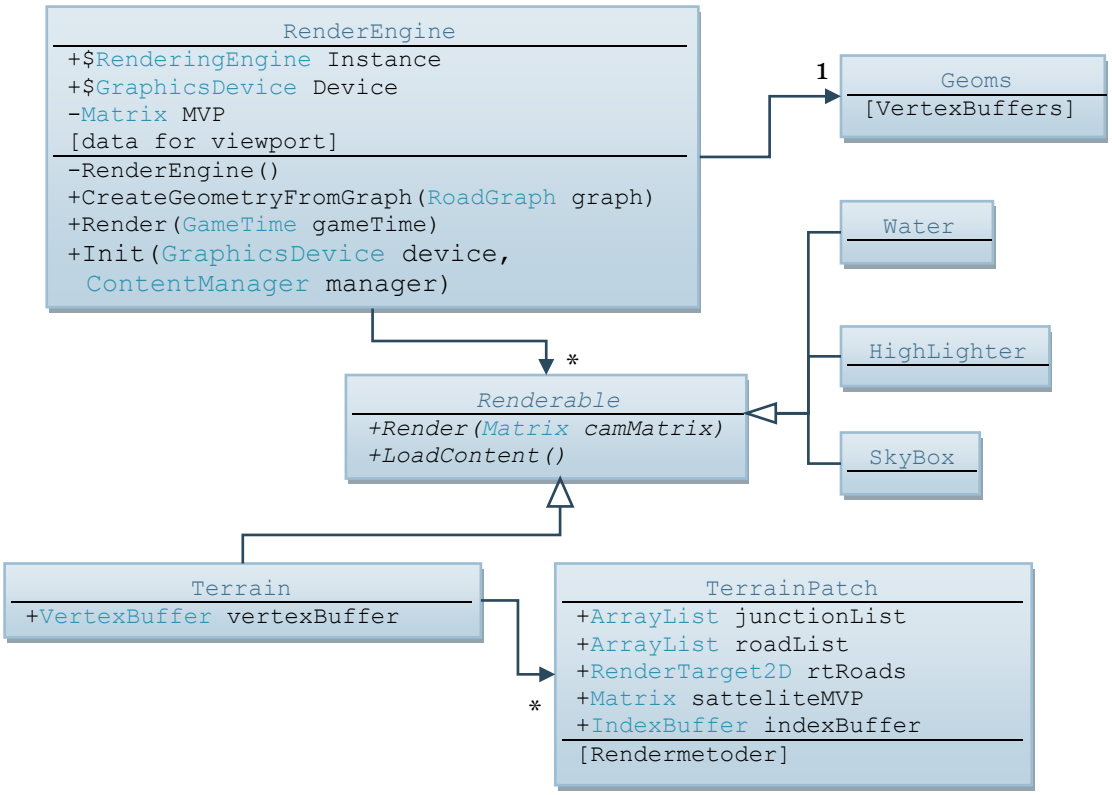
En af de vigtigste bestanddele i L-systemets omgivelser er den graf der repræsenterer det færdige vejnet. Når en vej skal justeres i forhold til de lokale omgivelser, spiller det eksisterende vejnet en vigtig rolle. *RoadGraph* er meget lig det geometriobjekt vi havde i *BoP*, og derfor har klassen en række metoder til manipulation af skildpadden.

Når en vej har fundet sin endelige orientering gemmes den, og eventuelle nye vejkryds gemmes i en rumlig datastruktur. Vi benytter et Quad-træ, der består af en række knuder der enten har 4 underknuder eller ingen (hvis det er et blad). Hver knude i grafen repræsenteres af en

QuadTreeNode, der består af en axis-aligned-bounding-box, *AABB*, som definerer knudens rumlige udbredelse, en liste af underknuder samt lister af de veje og kryds der er tilknyttet en specifik knude.

Veje og vejkryds repræsenteres af hhv. *Road* og *Junction*. Hver *Junction* har en *type* der f.eks. kan være `STREET` eller `HIGHWAY`. Af renderingshensyn har vi valgt også at lade vejkryds have en vejtype. Hvert vejkryds har en liste af de veje, der forbinder denne. Veje er geometrisk definerede ved en en *FiniteLine*, der udelukkende består af to vejkryds, og vi har således en fuldstændig dobbeltrettet graf, hvori alle veje er forbundne gennem vejkryds og alle kryds er forbundne gennem veje.

4.1.3 Grafikmotor



Figur 4-5 UML-diagram af grafikmotoren

Alle klasser, der skal indgå i grafikmotoren, nedarver fra den abstrakte klasse *Renderable*. *RenderEngine* ved på den måde hvilke funktioner, den skal kalde, når et grafikobjekt skal indlæses og tegnes. Ideelt skulle *RenderEngine* simpelthen have en liste med de *Renderable*s, der findes i scenen, og løbe dem igennem når de skal tegnes. Vi har dog brug for at kunne regulere, hvornår de forskellige objekter skal renderes. F.eks. skal vandet tegnes til sidst, da vi bruger framebufferen som refraktionstekstur, og vi skal ikke tegne det når refleksionsteksturen tegnes. Derfor har *RenderEngine* pointere til de enkelte objekter.

Vi har valgt at inddele vores terræn i felter, hvilket gør det muligt f.eks. at foretage frustumculling og frem for alt, at opdele vores vejtekstur (se afsnit 4.7) da teksturer på grafikkortet har en begrænset opløsning. *Terrain*-klassen indeholder derfor en enkelt `vertexbuffer`, der indeholder

alle punkter i terrænet, mens hver *TerrainPatch* har en `indexBuffer` der refererer til de punkter, der er specielle for det enkelte felt. Desuden har et felt lister over de veje og kryds indenfor feltets område, der bruges når feltet skal tegnes.

Til at tegne veje, fortov og vejkryds har vi brug for en række geometriske modeller. F.eks. kan vi beskrive en enkelt vej som et rektangel, der er skaleret, roteret og transformeret til den korrekte form og position. Mange dele af systemet har brug for disse modeller, og derfor har vi samlet dem i et objekt; *Geoms*. Vi holder en tilgængelig, statisk pointer til dette objekt i *RenderEngine*.

4.2 L-systemet

Vi bruger et L-system, der er meget lig det Müller bruger (i [11]). Der er enkelte mindre forskelle, som vi vil gennemgå i afsnit 4.7. Det kan ses herunder. Hovedtrækkene i L-systemets udformning er gennemgået i afsnit 2.2.

w: R(del, initialRuleAttr)?I(initRoadAttr, UNASSIGNED)

p_1: R(del, ruleAttr) : del<0 → ε

p_2: R(del, ruleAttr) > ?I(roadAttr, state) : state == SUCCEED
{globalGoals(ruleAttr, roadAttr)}
→ +(roadAttr.angle) F(roadAttr.length)
B(pDel[1],pRuleAttr[1],pRoadAttr[1])
B(pDel[2],pRuleAttr[2],pRoadAttr[2])
B(pDel[0],pRuleAttr[0],pRoadAttr[0])

p_3: R(del,ruleAttr)>?I(roadAttr,state) : state == FAILED → ε

p_4: B(del,ruleAttr,roadAttr) : delayConstraints(del, roadAttr)
== WAIT → B(del-1,ruleAttr,roadAttr)

p_5: B(del,ruleAttr,roadAttr) : delayConstraints(del,roadAttr)
== SPAWN → [R(del,ruleAttr) ?I(roadAttr,UNASSIGNED)]

p_6: B(del,ruleAttr,roadAttr) : delayConstraints(del,roadAttr)
== DEAD → ε

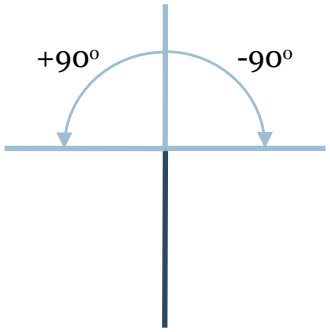
p_7: R(del,ruleAttr) < ?I(roadAttr,state) :
delayConstraints(del,roadAttr) == DEAD → ε

p_8: ?I(roadAttr,state) : state==UNASSIGNED
{localConstraints(roadAttr)}
→ ?I(roadAttr,state)

p_9: ?I(roadAttr,state) : state!=UNASSIGNED → ε

4.3 Globale mål

Som udgangspunkt indsætter L-systemet tre nye veje for hvert vejkryds, der forgrener sig som vist i Figur 4-6.



Figur 4-6 De tre ideelle forgreninger

Metoden *ApplyGlobalConstraints* vil efterfølgende modificere både vinkel og længde for de kommende veje. Vi skelner mellem forgreningerne, da den fremadrettede skal følge et anderledes mønster end de to andre.

Den første beslutning er, hvilken type den nye vej skal være. Den fremadrettede har altid samme vejtype som den foregående. I virkelige byer ser man praktisk taget aldrig, at f.eks. en gade ændrer sig til en hovedvej eller omvendt. Vi kunne have valgt, at det fremadrettede segment skiftede type med en yderst lav frekvens, men efter vores opfattelse vil dette ligne et visuelt artefakt snarere end bidrage til realismen.

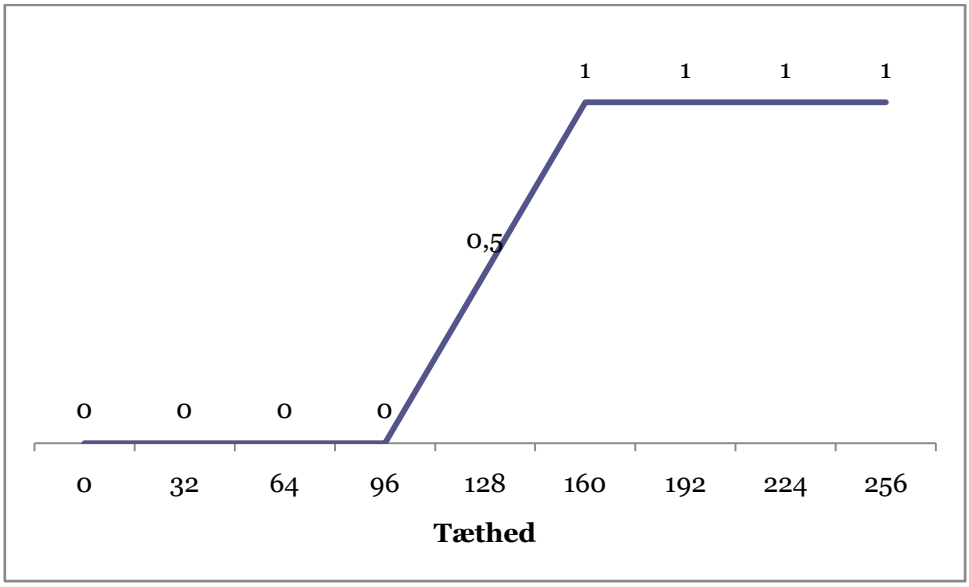
I vores implementering vil vi så vidt muligt bevare hovedvejene som ét sammenhængende net. Derfor kan gader ikke afføde hovedveje, og vi skal således kun afgøre type for forgreninger fra hovedveje. Dette foregår ved lodtrækning, hvor vægte for de enkelte typer sammen med befolkningstætheden på det pågældende punkt spiller ind.

4.3.1 Hovedveje

Når vi har afgjort, at en vej skal være en hovedvej, beregner vi indledningsvist hhv. rotation og vægt for de enkelte mønstre.

Manhattan-regel

Rotationen for *Manhattan*-reglen er den samme som for de ideelle forgreninger, hvilket giver et rektangulært mønster med rette vinkler i alle kryds. Vægten beregnes som funktion af befolkningstætheden. Brugeren har mulighed for at specificere en middelværdi hvor tætheden sættes til 0,5 samt en hældning hvormed vægten tiltager lineært (se Figur 4-7).



Figur 4-7 Vægt af Manhattanregel som funktion af befolkningstætheden (middelværdi 128)

Basic-regel

Rotationen for *basic*-reglen findes ved at undersøge både befolkningstæthed og terrænets hældning. Vi gennemløber en serie specificerede vinkler i forhold til forgreningsvinklen. Det kan f.eks. være fra -8° til +8° med et interval på 4°. For hver vinkel betragter vi en række punkter ad den nye akse med stigende afstand til udgangspunktet.

Til forskel for Pascal Müller [11] har vi ikke en speciel regel, der tager højdeforskelle i betragtning. Vi har i stedet valgt at indbygge denne effekt i *basic*-reglen. Heuristikken for et givent punkt, er derfor som følger:

```
(255-tæthed) + højdeforskel + punkthældning * 2
```

Vi trækker befolkningstætheden fra 255 (tætheden gemmes i et intensitetskort der har værdier mellem 0 og 255) og adderer højdeforskellen mellem udgangspunktet og det betragtede punkt, sammen med hældningen i det pågældende punkt. Jo højere værdi for det enkelte punkt, desto mindre attraktivt er det for vejnettet at bevæge sig i denne retning. Vi benytter punkthældningen, fordi veje derved søger mod områder der er jævne, hvor der således er basis for byudvikling.

Hvis positionen af det betragtede punkt befinder sig i vand, tillægges en ekstra høj heuristik, hvorved hovedveje kan svinge væk fra vand i god tid.

Vi finder hældningen som den maksimale højdeforskel mellem den betragtede celle og dennes naboer i et Von Neumann nabolag [21] (Figur 4-8).

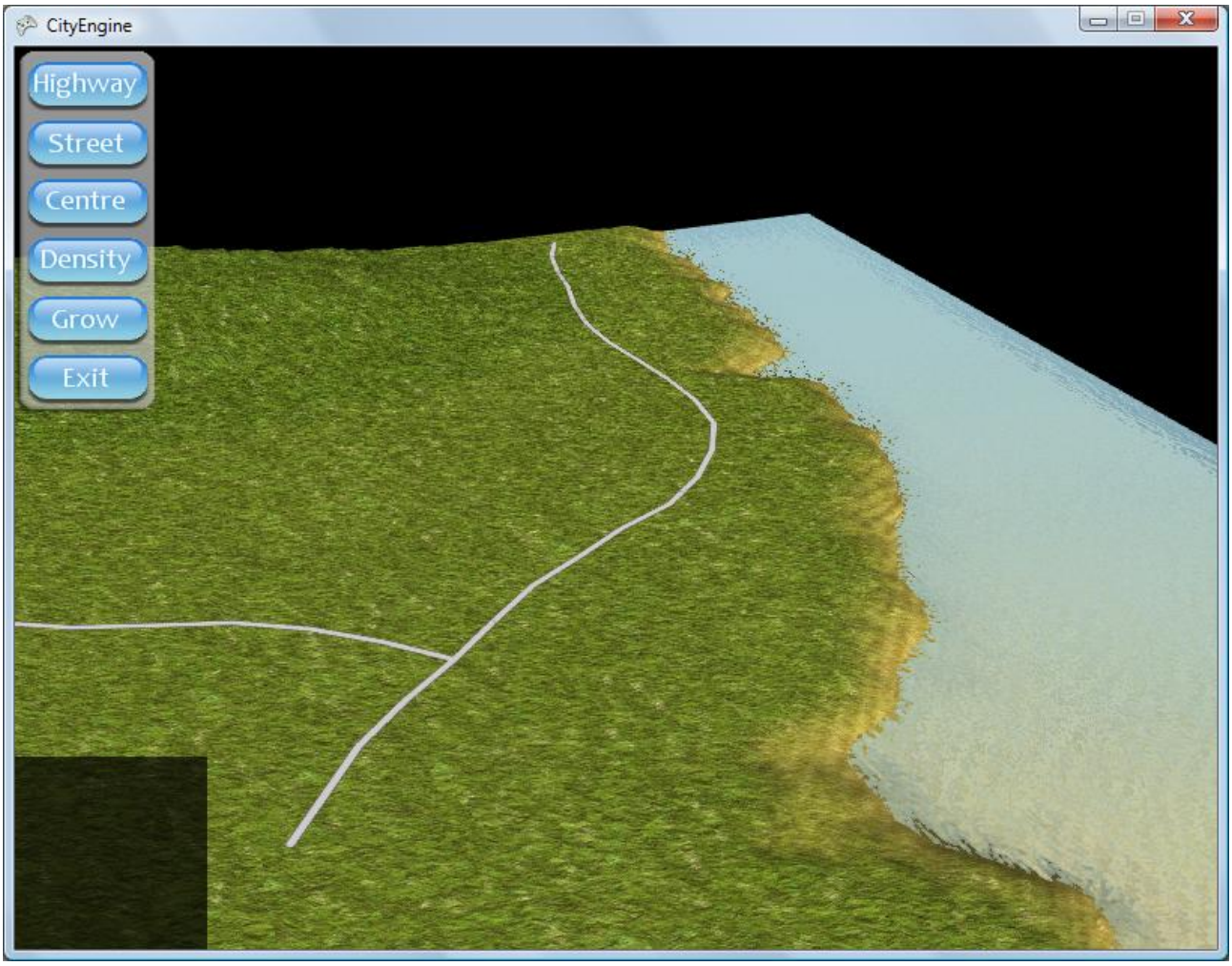
	i,j+1	
i-1,j	i,j	i+1,j
	i,j-1	

inc = max ((i,j) - i,j+1
(i,j) - i,j-1
(i,j) - i+1,j
(i,j) - i-1,j)

Figur 4-8 Beregning af punkthældning

Vi kan nu addere alle punktværdier for samme vinkel med vægtning svarende til den inverse afstand til udgangspunktet, og vi har således en heuristik for en given vinkel. Vi kan nu vælge den vinkel med laveste heuristik (hvis alle er ens vælges den fremadrettede).

Vi sætter i alle tilfælde vægten for *basic*-reglen til 1.0 (vi bruger 1.0 som referenceværdi for de øvrige regler).

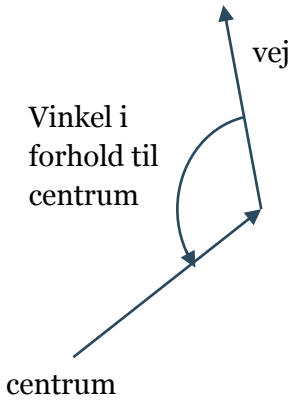


Figur 4-9 Hovedvej forsøger at undgå vand og kuperet terræn

Radial regel

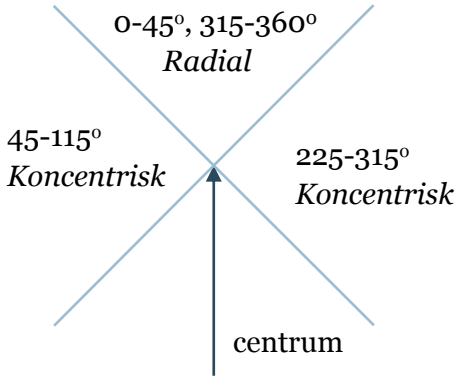
Beregningen af rotationen for den radiale regel baseres på brugerdefinerede bycentre. Vi har valgt fremgangsmåden med bycentre, fordi det i højere grad giver mulighed for at sammenkoble flere regler i samme by med forskellige lokaliteter.

Som udgangspunkt finder vi vejens vinkel i forhold til nærmeste centrum som vist i Figur 4-10.



Figur 4-10 Vinkel i forhold til centrum

Vi har nu følgende tilfælde:



Figur 4-11 Bestemmelse af en vejs retning

Ligger vinklen mellem 0-45° eller 315-360°, er vejen en radialvej, der fører trafik mod eller fra centrum af byen. Vi sætter vinklen til 0°. Ligger vinklen mellem 45 og 115° eller 225 og 315°, er vejen en ringvej, der fører trafikken udenom centrum. Vi sætter vinklen til hhv. 90° eller 270°. Peger vejen mod centrum foretager vi ingen tilpasning.

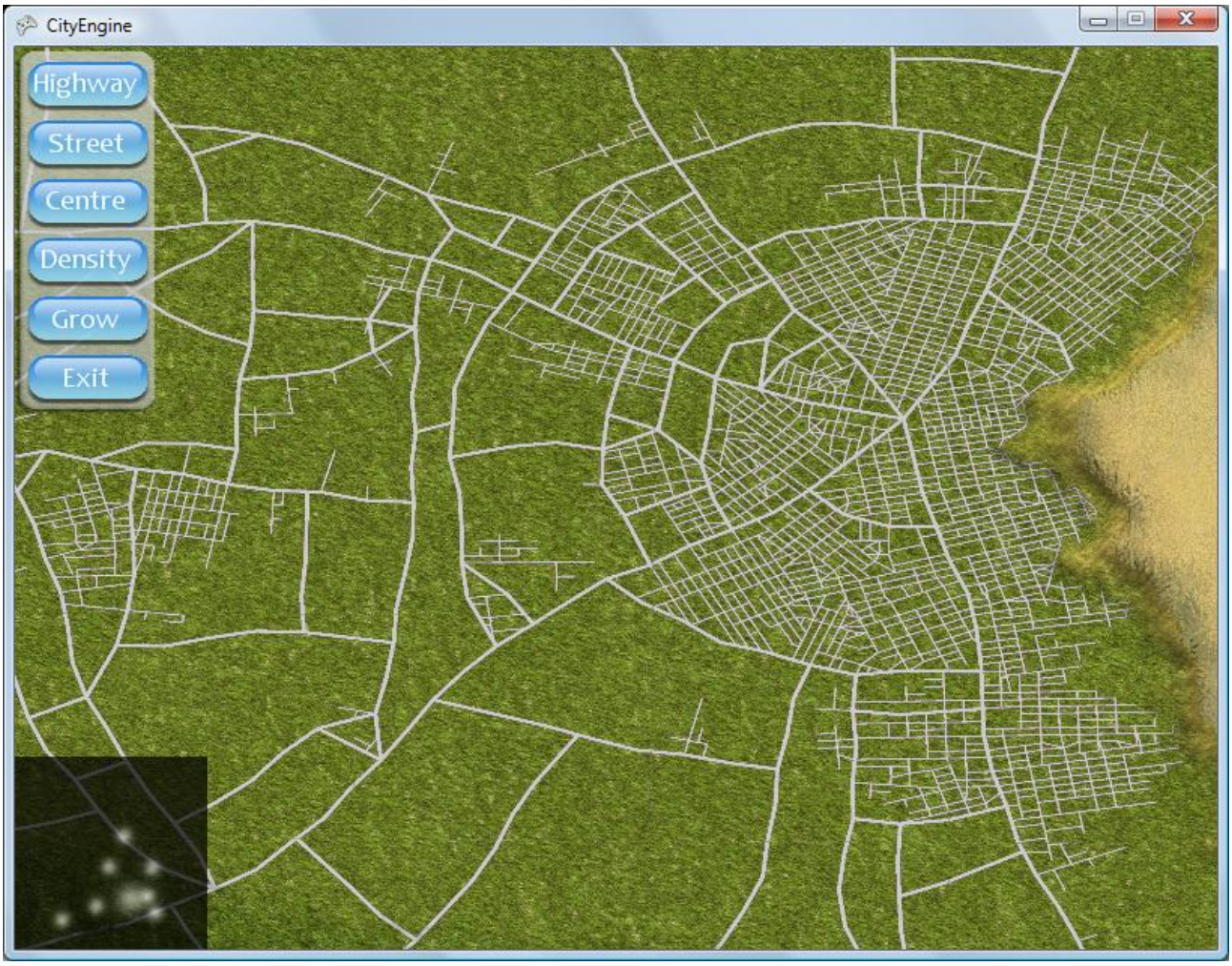
Vægten af radial-reglen beregnes som funktion af afstanden til det nærmeste centrum. Brugeren har mulighed for at knytte en radius til hvert centrum, og ud fra denne beregnes vægten som:

$$vægt = -(1 - invdist)^3 + 1$$

Hvor *invdist* er den inverterede afstand fra centrum, normaliseret i forhold til en brugerspecificeret maksimalafstand:

$$\text{invdist} = \max(radius - afstand) \,/ maxafstand$$

Både for den radiale regel og for *basic*-regelen kan vi opnå et mere realistisk visuelt udtryk ved, at tillægge vinklen en lille tilfældig faktor.



Figur 4-12 Radialregel, med ringveje og infaldsveje samt en mindre tilfældigheds faktor. Bemærk hvordan byens centrum let identificeres.

Blanding af regler

For anvendelsen af vores system er det helt afgørende, at en bruger kan blande de forskellige regler, og derved skabe unikke byer. Vores mål har været, at brugeren skal kunne modellere byen via nogle enkelte virkemidler: Man skal kunne indsætte centre, og tegne i tæthedskortet.

Byer, der følger den radiale regel, skal opstå i de områder hvor tætheden er høj og samtidigt nær ved et centrum. Byer der følger *Manhattan*-reglen, skal opstå i område med høj befolkningstæthed, men hvor der ikke findes centre i nærheden. I de områder hvor befolkningstætheden er lav, skal byen følge *basic*-regelen.

Indledningsvist finder vi rotationen og vægten for de tre regler. Vi trækker vægten af *radial*-reglen fra vægten af *Manhattan*-reglen (og *clamper* til 0). Dermed sikrer vi os at *radial*-reglen dominerer nær centre.

Vi finder den vægtede værdi af vinkler, ved først at beregne middelværdien af hhv. cosinus og sinus til vinklerne, og derefter rekonstruere vinklen:

$$middelCosinus = vægt * \cos(vinkel_a) + (1 - vægt) * \cos(vinkel_b)$$
$$middelSinus = vægt * \sin(vinkel_a) + (1 - vægt) * \sin(vinkel_b)$$

$$middelVinkel = tan^{-1}\left(\frac{middelSinus}{middelCosinus}\right)$$

Vi kan nu beregne middelvinklen mellem *Manhattan*-reglen og *basic*-reglen vægtet i forhold til *Manhattan*-reglen (*basic*-reglen er jo 1). Herefter kan vi finde middelvinklen mellem radial-vinklen og den nye vinkel, vægtet i forhold til *radial*-reglen. Dermed har vi en vinkel, der opfylder de ovenfor specificerede krav.

4.3.2 Gader

Pascal Müller argumenterer i sin artikel [11] for, at gader former rektangulære blokke ud fra observationer af virkelige byer. Vi mener desuden, at den rektangulære struktur har en meget afgørende indflydelse på det visuelle udtryk. Betragter man gaderne i Figur 4-7, ses det tydeligt, at forskelle i blok orienteringen gør de enkelte kvarterer identificerbare, hvilket ikke ville være synligt ved kvadratiske blokke.

Vi implementerer blokke, ved simpelthen at markere de enkelte forgreningers *ruleAttr* med en indikator, der fortæller om gaden ligger i gavlen eller siden af blokken. Forgreninger af en vej sættes til det modsatte af den gamle vej, mens fremadrettede veje beholder samme type. Herefter sættes vejens længde til brugerdefinerede værdier for højde og bredde af en blok.

Efter at hhv. en gade eller en hovedvej er initialiseret, kontrollerer vi hældningen af den foreslåede vej, og hvis denne overstiger et givet maksimum, fjernes vejen fra L-systemet. Dette gøres ved at sætte dens *delay*-værdi til mindre end 1, således at modulet fjernes af en senere produktionsregel (p_6) i L-systemet.

4.4 Lokale begrænsninger

Når en vej er blevet tildelt globale mål, skal den efterfølgende rettes ind efter de lokale omgivelser. Vi undersøger for de samme forhold som Pascal Müller gør, dog med én afgørende forskel. I hans artikel [11] dannes hovedvejene før gaderne, mens vi indsætter begge dele nogenlunde samtidigt. Da vi betragter de mellemliggende skridt som ligestillede med den færdige by, skal det visuelle indtryk også her være overbevisende. Man kan for eksempel ikke forstille sig en by udelukkende bestående af hovedveje. For at føje yderlige til realismen, har de forskellige nye veje varierede forsinkelser. Vi kan derfor risikere et scenarie som i Figur 4-13, hvor to kvarterer mødes.

Figur 4-13 Sammenstød mellem to kvarterer

Dette er yderst sjældent med Müllers system[11], da hovedvejene normalt afgrænser et helt kvarter. Hvis ikke en vej kender alle andre veje, inklusivt de veje der er indsat i samme afledning af L-systemet, får vi et resultat som i Figur 4-14, hvor tydelige artefakter fremstår fordi nye veje ikke finder hinanden.

Figur 4-14 Sammenstød mellem to kvarterer, hvis vejene ikke kender de i samme afledning indsatte

Vi kan altså ikke indsætte vejene i et omgivelses-skridt i L-systemet. I vores implementering indsætter vi vejene umiddelbart efter, at have behandlet de lokale begrænsninger, da vi her kender vejens endelige position og orientering.

Korrektion for vand

Ved undersøgelser af de lokale omgivelser, undersøger vi først om vejen ender i vand. En vej vil, hvis den følger *basic*-reglen, forsøge at undvige vand, der ligger foran den. Dette forhindrer dog ikke vejen i at ramme vandet, og her træder de lokale begrænsninger i kraft.

For alle typer af veje forsøger vi først at rotere vejen til en position, så den ikke længere rammer vand. Vi gør dette på samme måde, som da vi forsøgte at optimere vinkelheuristikken for *basic*-reglen.

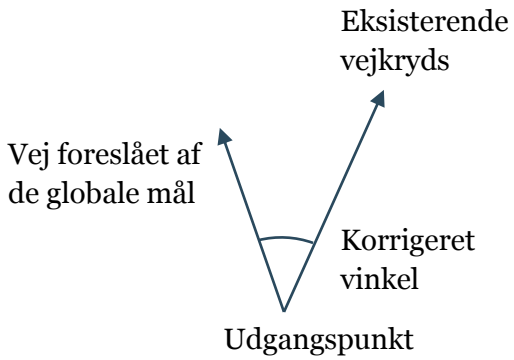
Kan det ikke lade sig gøre at rotere vejen til en position, hvor den ender på land, forsøger vi at afkorte vejen. I vores implementering halverer vi simpelthen vejlængden, og undersøger for om den stadig er i vand.

Hvis ingen af de to ovenstående metoder kan anvendes, sættes vejens `state` til *FAILED*, og den vil efterfølgende blive fjernet fra L-systemet.

Selvsensitivt L-system

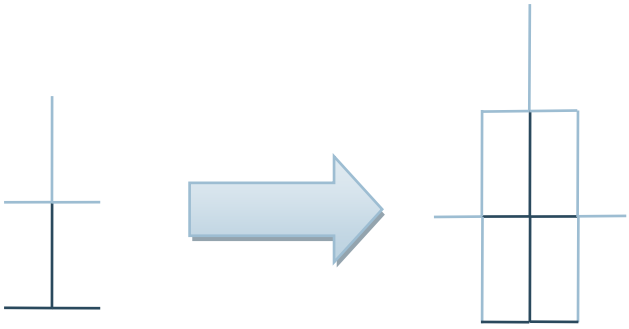
Hvis vejen overlever korrektionen for vand tester vi for, om den skærer andre veje. Er dette tilfældet, afkorter vi vejen. I afsnit 4.5 vil vi gennemgå den egentlige skærings-algoritme.

Efterfølgende søger vi efter, om der findes et allerede eksisterende kryds i endepunktets umiddelbare nærhed. Er dette tilfældet, korrigerer vi vejens rotation og længde, så den rammer det pågældende vejkryds. Den korrigerede vinkel beregnes, ved at betragte den ønskede vej og den oprindelige vej som vektorer, og deraf finde cosinus til vinklen (de to vektorer normaliseres, sådan at prikproduktet er cosinus til vinklen). Vi ganger med krydsproduktet, for at få det korrekte fortegn. Det er vigtigt, at vi ikke forsøger at korrigere, hvis vejen faktisk ender i det tiltænkte punkt, da krydsproduktet her ikke er defineret.



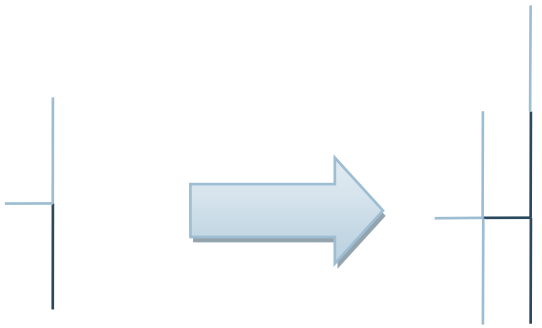
Figur 4-15 Beregning af vinkel til nærmeste eksisterende vejkryds

Da vi, til forskel for Müller, har variation i forsinkelserne af veje, opstår der et nyt problem når vi søger efter den nærmeste nabo. I Müllers system er der ingen forsinkelse i forgreningen af veje, og opbygningen af en blokstruktur foregår som i Figur 4-16.



Figur 4-16 Ideel opbygning af blokstruktur

På grund af variationen i forsinkelserne er vi ikke sikret denne fremgangsmåde. I vores implementering opstår situationen i Figur 4-17 ofte.



Figur 4-17 Opbygning af blokstruktur med varierede forsinkelser

Når vi søger efter den nærmeste nabo i scenariet fra Figur 4-17, opstår der et problem, når vi skal afgøre i hvor stor radius, vi skal søge efter den nærmeste nabo. Søger vi i for stor radius, vil vi finde naboen, i den parallelle vej i samme blok (Figur 4-18).

Figur 4-18 For stor søgeradius

Dette vil forårsage tydelige og uønskede artefakter i blokstrukturen (Figur 4-19).

Figur 4-19 Blokstruktur-artefakt

Der findes ingen elegant løsning på problemet. I vores implementering søger vi i en radius svarende til lige under en halv gang af vejlængden for gader, og samtidig sørger vi for at længde af gavlen ikke er mindre end halvdelen af længden af siden. Dette giver efter vores mening et tilfredsstillende resultat, og artefaktet bliver praktisk taget elimineret.

I vores implementering søger hovedveje ikke efter nabokryds, der udelukkende er forbundet med gader. Vi har fundet, at dette giver et pænere resultat, og dette er således endnu en konsekvens af at hovedveje og gader dannes på samme tid.

Hvis vi hverken finder en skæring med andre veje eller et eksisterende vejkryds i nærheden, forsøger vi at finde en skæring længere fremme, nøjagtigt som i Müllers artikel.

Rækkefølgen af de ovenstående skridt er vigtig. Forestiller vi os, at vi først søgte efter den nærmeste nabo, og herefter for skæring af andre veje, vil vi ofte få kryds, der ligger meget tæt på hinanden. Det skyldes at vi, hvis vi forkorter en vej, ikke har muligheden for at finde et nabokryds til vejens nye position.

Vi stopper vejens forgrening, når den rammer dele af det eksisterende L-system. Dertil har vi indført en ny mulig `state`, *FINALIZED*, der sørger for at vejen indsættes, men at L-systemet ikke indsætter forgreninger i forlængelse af modulet.

Når vi har rettet vejen ind efter de lokale begrænsninger, indsættes den i grafen. Har vi f.eks. fundet et nabokryds, tilføjes den nye vej hertil, og den nye vejs slutkryds sættes som nabokrydset. Hvert vejkryds er altså unikt for det givne punkt, og vi kan dermed undersøge, om vejen findes i forvejen, ved at søge alle krydsets veje igennem. Hvis vi finder en vej, der har samme start og slutpunkt, indsættes den nye vej ikke, og dens `state` sættes til *FAILED*. L-systemet vil dermed

fjerne modulet, der definerer vejen. Dermed sikrer vi os, at der ikke findes for mange overflødige veje både i L-systemet og i grafen.

4.5 Rumlig datastruktur og skæring

Når byen vokser, får vi brug for at fortage rigtigt mange skærings-beregninger. Ved at holde vejnettet i to dimensioner, opnår vi en stor optimering. For det første, skal en mindre mængde data behandles, og mange af de skæringsalgoritmer vi bruger, bliver meget simple. Vi har valgt at optimere dette led i systemet, ved at implementere en rumlig datastruktur. Vi har brug for en struktur, hvor man hurtigt kan uddrage og indsætte data.

Et kD-træ giver en rigtigt hurtig søgetid, mens indsættelsestiden kan blive meget langsom, da vi ofte har brug for at balancere træet.

Vi har valgt et quad-træ, da vi her får en rimelig optimering i søgetid, samtidig med at arbejdet ved indsættelse af data bliver meget begrænset.

Vi bruger følgende skæringsalgoritmer (alle i 2-dimensioner)

Vej-kvadrat skæring

Vi har brug for at finde skæringen mellem en vej og knude i quad-træet. Knuden definerer et kvadrats max og min-punkt (i AABB). Algoritmen er således en *ray-box* skæring [22], vi har modificeret en anelse for at tage højde for at veje er endelige.

$$t1 = \frac{min - p1}{dir}$$

$$t2 = \frac{max - p1}{dir}$$

$$indgang = \max(\min(t1.i, t2.i), \min(t1.j, t2.j))$$

$$udgang = min(\max(t1.i, t2.i), \max(t1.j, t2.j))$$

Hvor *dir* er vektoren fra vejens start- til vejens slut-punkt. *Da* vi dividerer igennem med *dir*, har vi en skæring hvis $indgang < udgang$ samtidig med at $indgang < 1$ og $udgang > 0$ (da vi jo har normaliseret i forhold til vejen længde).

Vej-vej skæring

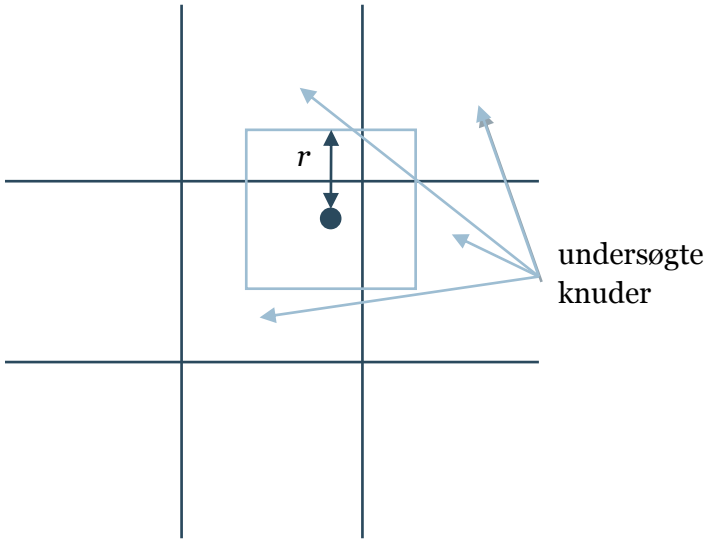
Vi har brug for at finde skæring mellem to veje, når vi undersøger for de lokale begrænsninger. Vi benytter algoritmen fra *Real-Time Rendering* [22] side 617.

Nærmeste nabo

Til at finde det nærmeste vejkryds til et punkt benytter vi vores quad-træ. Når vi besøger en knude i træet, undersøger vi alle vejkryds og gemmer den nærmeste (hvis der findes et kryds, der er nærmere end de kryds, vi har undersøgt i andre knuder). Hver underknude undersøges hvis følgende gælder:

$$\begin{aligned} &pos.i - q.min.i + radius \geq 0 \\ \&\&\ &q.max.i - pos.i + radius > 0 \\ \&\&\ &pos.j - q.min.j + radius \geq 0 \\ \&\&\ &q.max.j - pos.j + radius > 0 \end{aligned}$$

Hvor *q* er den undersøgte knude, og *radius* er den hidtil korteste afstand. Vi bruger altså *radius* til at definere et kvadrat, hvori alle skærende knuder undersøges (Figur 4-20)



Figur 4-20 Søgen efter nærmeste nabo

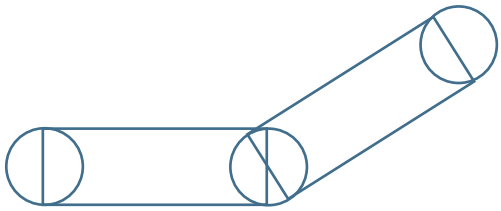
Vi vil redegøre for algoritmens rigtighed i testafsnittet.

4.6 Visualisering

Visualisering af vejnettet varetages af *RenderEngine*, der tager en graf som argument til metoden `CreateGeometryFromGraph`.

Vi beregner en rotationsmatrix, en translationsmatrix og en skaleringsmatrix til hver vej i grafen. Rotationen og translationen bestemmes af vejens position og orientering, mens skaleringen bestemmes af vejens længde. Vi kan nu rendere ét generisk vejmodul til alle veje, der er transformeret ind i den enkelte vejs position og form. Vi har til fortov brug for, at kunne lægge teksturer på vejmodulet, og derfor skalerer vi også teksturkoordinaterne, og belægger fortovet med en *tileable* tekstur. Vi har valgt denne fremgangsmåde, fordi vi derved ikke behøver at holde en stor mængde geometri på grafikkortet. Dette gør dog renderingen langsommere, men i vores tilfælde er dette ikke et stort problem. Vi vil tegne vejnettet til en tekstur, og selve renderingen af vejsegmenterne sker dermed med en meget lav frekvens (når L-systemet udvides), og vi har mere brug for video-hukommelse til pænere teksturer.

Vi baserer tegningen af vejene på følgende princip illustreret i Figur 4-21.

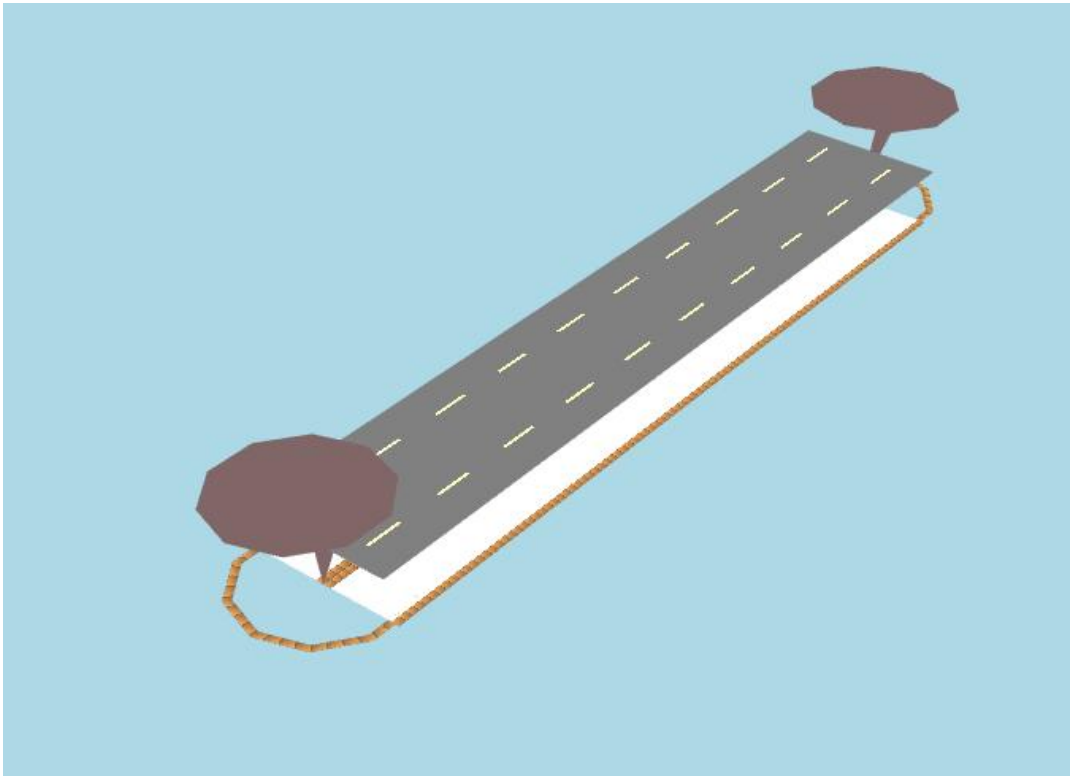


Figur 4-21 Tegning af vejsegmenter og vejkryds

Vejkryds tegnes som en skive med diameter svarende til vejenes bredde. Vejene tegnes som rektangler, der har udgangspunkt i et givet vejkryds, og er roterede i forhold til samme vejkryds. Når polygonerne fyldes ses kryds ikke.

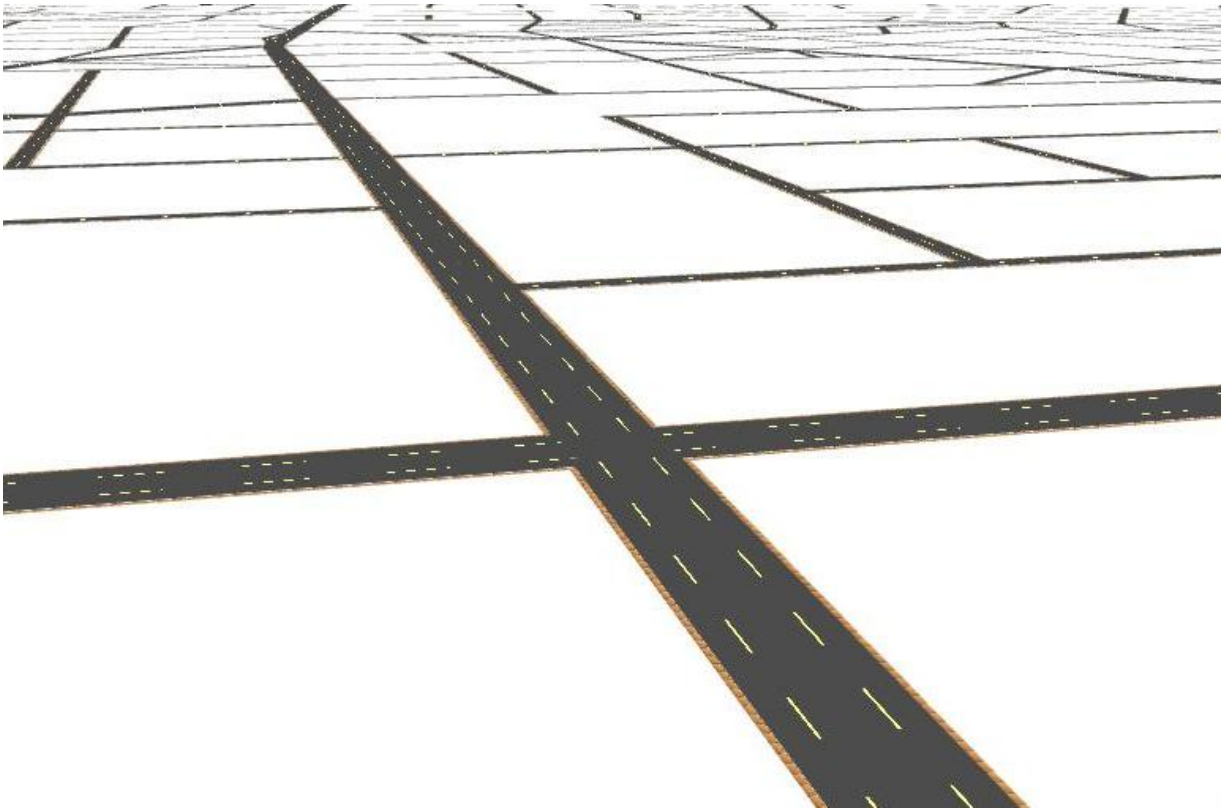
Figur 4-22 Fyldte vejkryds og vejsegmenter

Det er klart, at denne metode ikke tager højde for fortov, da disse ved vejkryds vil gå tværs over vejen. Vi tegner fortov som polygoner, der er en anelse bredere end "fyldet". For at undgå at fortovene krydser vejene, tegner vi de enkelte dele på følgende måde:



Figur 4-23 Et vejsegment renderet med stor forskydning i lag

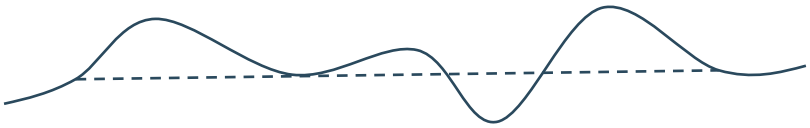
De forskellige dele renderes i lag. Fortovene tegnes først, hvorefter vejene tegnes. De dele af fortovet der krydser veje bliver således skjulte. Det endelige resultat af visualiseringen ses i Figur 4-24



Figur 4-24 Visualisering af vejnet

4.7 Vejnet i terræn

Vores system tager højdekortet i betragtning, når det genererer vejnettet. Det er derfor vigtigt, at vi også kan tegne byen i terrænet. Vi kunne vælge at tegne vores veje som polygoner, der roteres så start og slutpunkt er i terrænets højde. Problemet med denne metode er illustreret i Figur 4-25



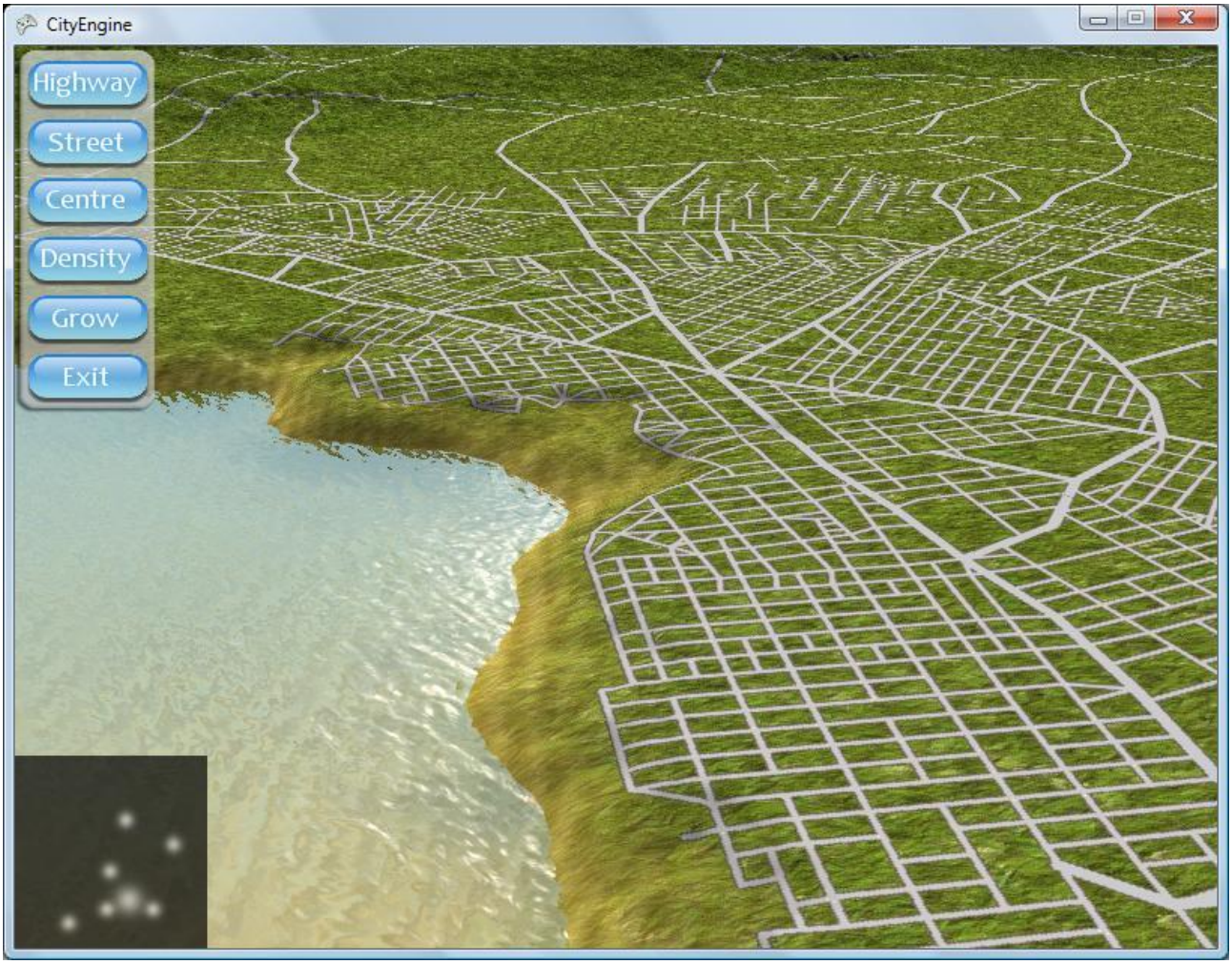
Figur 4-25 Prøblem med terræn

Vejen overlappes af terrænet. Vi kan løse dette problem, ved at rette terrænet til, som vist i Figur 4-26

Figur 4-26 Løsning på førnævnte problem

Vi har valgt en løsning, der baserer sig på at tegne vejnettet til en tekstur, og projicere denne ned på terrænet. Problemet med denne metode er, at vi har brug for meget store teksturer for at eliminere de værste aliasing problemer. Der er en grænse for hvor store enkelte teksturer man kan sende til grafikkortet, og vi er derfor nødt til at indele disse i mindre felter. Vi tegner til en tekstur hvor alfabufferen er sat til 0 sådan, at terrænet træder frem der, hvor der ikke er veje.

Det endelige resultat ses i Figur 4-27



Figur 4-27 Illustration af vores valg

Det er tydeligt, at vi ikke har den samme detaljegrad ved brug af teksturer, som vi havde da vi rendererde polygoner. Dette skyldes den begrænsede opløsning i teksturer.

4.8 Globale variable

Når en vej forgrener sig, skal vi tage stilling til hvilken type, der skal instantieres. Vi fortager dette valg ved at evaluere et tilfældigt tal (c) med følgende heuristik:

$$c < typeChance + befolkningstæthed * typeChanceModifier$$

Her er `typeChance` og `typeChanceModifier` specielle variable, der er defineret for hver type vej. Vi kan altså styre en statisk parameter sammen med en parameter, der er afgjort af tæthed. På den måde kan vi f.eks. vælge at chancen for en hovedvej er større i områder med høj befolkningstæthed.

Som det fremgår har vi rigtigt mange parametre til at definere vores by. Vi har samlet alle relevante variable i klassen *GlobalVariables*.

Man kan måske argumentere for, at der er alt for mange variable at skrue på, men vi mener, at det ikke er et stort problem, netop i vores tilfælde. Vi forestiller os ikke at almindelige brugere har adgang til disse variable, men at et underliggende system, der f.eks. skal illustrere en middelalderby, håndterer brugerens input og sætter forskellige parametre.

5 Egne idéer og tilføjelser

Dette afsnit omhandler de udvidelser, vi har lavet til Pascal Müllers arbejde, og hvordan vi har implementeret dem. Vi har arbejdet videre på hans L-system (fra [11]), og er kommet frem til det, der ses herunder.

w: $R(del, initialRuleAttr)?I(initRoadAttr, UNASSIGNED)$

p_1: $R(del, ruleAttr) : del<0 \rightarrow \varepsilon$

p_2: $R(del, ruleAttr) > ?I(roadAttr, state) : state == SUCCEED$
$\{globalGoals(ruleAttr, roadAttr)\}$
$\rightarrow +(roadAttr.angle)\ \ F(roadAttr.length)$
$B(pDel[1],pRuleAttr[1],pRoadAttr[1])$
$B(pDel[2],pRuleAttr[2],pRoadAttr[2])$
$B(pDel[0],pRuleAttr[0],pRoadAttr[0])$

p_3: $R(del,ruleAttr)>?I(roadAttr,state) : state == FAILED \rightarrow \varepsilon$

p_4: $B(del,ruleAttr,roadAttr) : delayConstraints(del,roadAttr)$
$== WAIT \rightarrow B(del-1,ruleAttr,roadAttr)$

p_5: $B(del,ruleAttr,roadAttr) : delayConstraints(del,roadAttr)$
$== SPAWN \rightarrow [R(del,ruleAttr)\ ?I(roadAttr,UNASSIGNED)]$

p_6: $B(del,ruleAttr,roadAttr) : delayConstraints(del,roadAttr)$
$== DEAD \rightarrow \varepsilon$

p_7: $R(del,ruleAttr) < ?I(roadAttr,state) :$
$delayConstraints(del,roadAttr) == DEAD \rightarrow \varepsilon$

p_8: $?I(roadAttr,state) : state==UNASIGNED$
$\{localConstraints(roadAttr)\}$
$\rightarrow ?I(roadAttr,state)$

p_9: $?I(roadAttr,state) : state!=UNASSIGNED \rightarrow \varepsilon$

I forhold til Pascal Müllers er det næsten det samme, dog med et par ændringer. I p_2 har vi ændret et vejmodul til et forgreningsmodul, hvilket gør, at vi kan opnå forsinkelse på de fremadrettede veje ligesom forgreningerne. Vi har desuden lavet *delayConstraints*-funktionen, der håndterer den udvidede måde, hvorpå vi bruger forsinkelse som beskrevet nedenfor.

5.1 Mål for udvidelser

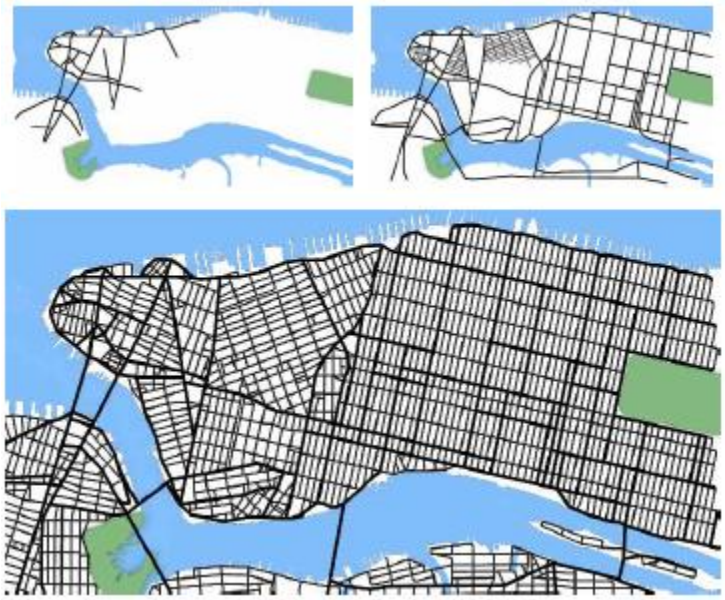
Vi har ønsket at gøre bygenereringen mere interaktiv, og give brugeren mere kontrol over resultatet. Det skal være muligt at påvirke byen, og ændre dens udseende efter behov, mens den vokser.

Vi vil gøre det muligt at indsætte flere befolkningszoner på landskabet. Disse byer skal selv vokse frem hvor brugeren ønsker det, og automatisk danne deres eget vejnet. Byerne skal fremkomme når det eksisterende vejnet når hen til den ønskede position for den nye by. Man kan så hjælpe den på vej ved selv at sætte veje ind med en rigtige retning.

Derudover vil vi gøre det muligt at ændre på eksisterende byer mens de vokser, og gøre denne proces mere dynamisk. Dette kan gøres ved at indikere hvor byen skal vokse hen, enten ved at fortælle programmet at byen skal udvikle sig et givet sted hen, eller ved selv at indsætte en eller flere vej, som byen så bruger i sin videre udvikling.

5.2 Udvidet brug af forsinkelse i L-systemet

Pascal Müller bruger forsinkelser til at få den rette struktur i sine byer. Forsinkelserne skal først og fremmest sørge for, at nettet af hovedveje skabes før gader udfylder rummene mellem disse. Denne effekt ses i Figur 5-1.



Figur 5-1 Fra Pascal Müllers test for Manhattan, visualiseret i 3 trin.

Vores idé baserer sig på at bruge disse forsinkelser til at gøre systemet interaktivt.

”Hvordan kan forsinkelser gøre systemet interaktivt?”:

Det første problem vi støder på når byen skal gøres levende er, at de mellemlæggende trin ikke virker overbevisende. Figur 5-1 gør det klart, at kun det endelige produkt ser naturligt ud, da man f.eks. ikke kan forestille sig en by udelukkende bestående af hovedveje (Figur 5-1 øverst til venstre). En mere sofistikeret brug af forsinkelser giver en meget overbevisende effekt. Vi kan f.eks. indføre konventioner om, at gader forsinkes mindre, når de er en forgrening af en hovedvej, og fremadrettede hovedveje en mindre forsinkede end forgrenede hovedveje.

Vi kan forbedre denne effekt markant ved, at gøre forsinkelserne afhængige af befolkningstætheden. I de zoner hvor tætheden er høj, vokser byen generelt hurtigere end i zoner med lav tæthed. Vi er selvfølgeligt stadig nødt til at differentiere mellem forskellige typer af veje og forgreninger, så byen vokser på en realistisk måde.

Vi har nu et interaktivt L-system. Vi kan ændre intensiteten i områder, og byens vækst vil reflektere dette. Ved at tillægge forsinkelserne en tilfældig faktor, virker interaktionen naturlig. Problemet er, at vi ikke kan påvirke veje, der indledningsvis er blevet tildelt en meget høj forsinkelse. Vi kan f.eks. have følgende situation: En vej venter med *del* = 200, og vi øger tætheden markant i vejens nærområde. I dette tilfælde vil den enkelte vej ikke blive påvirket, og vi skal altså vente i 200 iterationer af L-systemet før byen kan vokse videre. De nye veje vil efterfølgende vokse med den korrekte hastighed, der afspejler den højere tæthed.

For at tage hånd om det førnævnte problem, har vi omdefineret vores metode til forsinkelser. I stedet for at lade forsinkelserne afhænge af befolkningstætheden, sættes *del* til en særlig værdi specificeret for de enkelte vejtyper og forgreninger, og lige som før tillægges en faktor af tilfældighed. Derimod lader vi tidspunktet for instantieringen afhænge af tætheden i endepunktet af den pågældende vej. Hver gang en vejs forsinkelse evalueres i L-systemet, sammenlignes værdien med befolkningstætheden, og er *del* lavere end tætheden, instantieres vejen. En ændring i tæthedskortet vil således have umiddelbar effekt på de ventende veje.

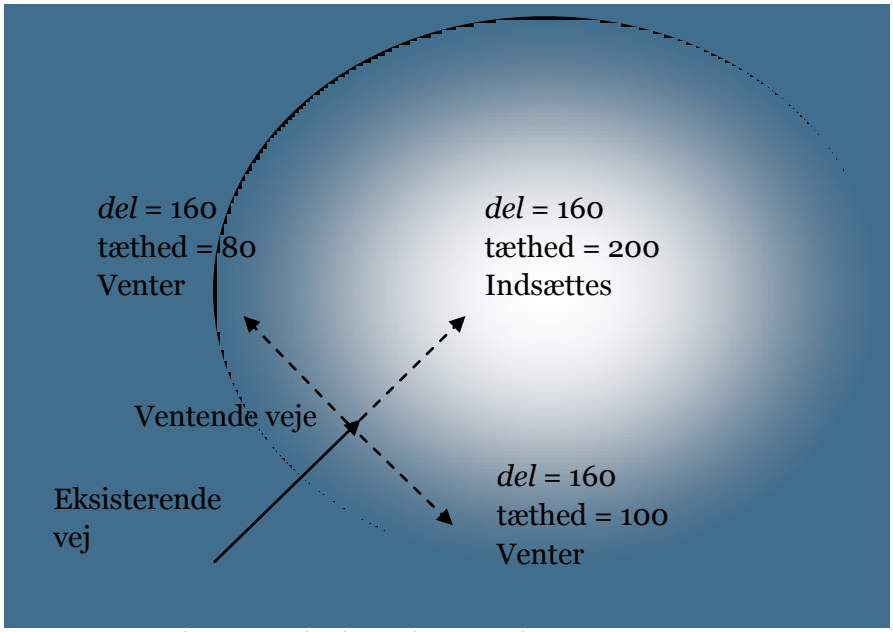


Figure 3 Princippet i det forsinkede L-system.
Kun den fremadrettede vej bliver indsat i den efterfølgende iteration. De to andre venter.

Evalueringen af forsinkelser varetages af metoden *delayConstraints*, der kaldes når reglerne; p_4, p_5, p_6 og p_7, undersøger deres betingelser. Metoden kan returnere *SPAWN*, *WAIT* eller *DEAD*, hvis vejen hhv. skal indsættes, vente eller slettes. En vej skal udgå fra L-systemet i det tilfælde hvor *del* er mindre end 0, nøjagtigt som hos Müller. Det er fortsat p_4, der varetager nedtællingen af *del*.

Ved forsøg har det vist sig, at interaktionen bliver mest realistisk, hvis gader ikke bliver indsat før tætheden er større end 0. Dette forårsager dog et artefakt, idet man ofte vil opleve at kun gader og ingen hovedveje, vil vokse frem når man øger befolkningstætheden fra 0 (nettet af hovedveje er fuldt udvokset). Vi har derfor indført en ny vejtype kaldet *avenue*, der på alle punkter svarer til en

hovedvej, bortset fra en den kun får lov til at vokse, når tætheden er større end 0. Vi vil på den måde have både latente hovedveje (avenuer) og gader fordelt i de ikke befolkede zoner.

5.3 Indsættelse af veje i L-systemet

Vi har gjort det muligt at indsætte nye veje i L-systemet, mens det udvikler sig. Rent praktisk gøres dette ved, at vi registrerer hvor brugeren har ønsket at placere vejen, og ændrer L-systemet til at tage højde for denne vej. Vi har måttet tilføje et nyt modul til L-systemet, som vi kalder *Identity*. *Identity* går tilbage til udgangspunktet, og translaterer derefter hen til den ønskede position. Dette gør at vi, ved indsættelse af vej, bare kan tilføje en række moduler til L-systemet. Vi indsætter et *Identity*-modul for at komme hen til det rigtige sted for den nye vej. Derefter kommer der, som i produktionsregel p_2, et rotationsmodul(+) efterfulgt af et *F*-, et *R*- og et *?I* -modul. Dette omsluttes af et push- og et pop-modul, for at L-systemet kan bruge det i dets videre voksen.

Hvis den indsatte vej starter i et eksisterende vejkryds, har vi valgt at stoppe væksten af de andre veje, der mødes i dette kryds. Dette gøres da der, ellers kan opstå fejl med veje der ligger meget tæt, eller vejkryds med alt for mange veje. Vi lader et vejkryds vide, om det er blokeret eller ej, og sætter så denne værdi, når vi indsætter en ny vej.

Denne mulighed for at indsætte veje, gør det samtidigt muligt at lave sin egen startby. Man kan sætte store og små veje ind på landskabet, og derefter lade byen vokse frem fra dette. Dette giver mulighed for at opnå meget realistiske byer, hvis man for eksempel tegner et gammelt middelalderligt bycentrum. Når byen så vokser ud fra dette centrum, vil den danne en mere moderne udseende by omkring, hvilket giver en meget virkelighedstro by. Mange gamle byers centre er nemlig, i høj grad, præget af hvordan byen så ud i middelalderen.

Et problem ved Pascal Müllers metode, og brugen af tilfældighed, er at der kan opstå tomme områder. Områder omkring et bycentrum, hvor der ikke er nogle små veje, men kun store. Dette kan give effekten af parker og lignende afgrænsede områder, men kan også se mærkeligt ud hvis der er for mange af dem. Dette kan undgås, ved selv at hjælpe byen på vej. Der indsættes bare en gade i dette område, og L-systemet vil så fylde det ud afsig selv. Se testafsnittet for illustration af dette.

6 Test og resultater

Dette bliver ikke et traditionelt software-testafsnit, da det ikke rigtigt er muligt med vores projekt. Projektet er ikke fejlfrit, da det er et ”proof-of-concept”-projekt, og kun skal illustrere at vores idéer og metoder fungerer. Man kan ikke forvente at kunne genskabe en by helt nøjagtigt, da der er en del tilfældigheder indblandet. Dette gør det meget svært at udføre en grundig strukturel test, og vi har derfor lavet en række gennemløb af programmet og set, at en ønsket effekt er konsekvent i sin optræden.

På baggrund af dette, omhandler vores testafsnit de ting vi har implementeret, hvordan de fungerer og hvordan de influerer byerne, samt det visuelle aspekt af projektet med hensyn til hvor realistiske byer vi kan frembringe. Afsnittet er delt op i fire dele, der gennemgår forskellige områder af projektet, og illustrerer hvad vi har opnået. Først er der en lille systematisk test af to forskellige krydsnings-algoritmer, for at sikre os at de giver det rigtige resultat i alle tilfælde. Dernæst en illustration af hvordan de forskellige regler influerer på byerne, og hvordan man kan kombinere disse regler. Efter dette vises hvordan det fungerer når man manipulerer L-systemet, og dermed ændrer på byen mens den vokser. Til sidst vil vi sammenligne med et virkeligt eksempel, og se om vi med rette kan påstå at det er en realistisk modellering af et vejnet.

Vi har valgt ikke at teste vores program til generering af træer, da det primært blev lavet for at forstå L-systemer i dybden. Vi er klar over at programmet ikke kan håndtere forkert udformede input-XML-filer, men har valgt ikke at gøre noget ved det, og koncentrere os om byprogrammet i stedet for.

6.1 Datatests

Vi har lavet to algoritmer, der skal testes strukturelt: Den måde vi undersøger for krydsning mellem to vejstykker, og den måde vi undersøger for om et vejstykke ligger i en given node i vores quad-træ.

Krydsning mellem to vejstykker

Vi bruger her algoritmen fra [22] som nævnt i afsnit 4.5. Vi opstiller de tre mulige scenarier, og undersøger om algoritmen giver det rigtige resultat.

De tre tilfælde er:

- De to vejstykker krydser hinanden
- De to vejstykker er parallelle
- De to vejstykker ville krydse hvis de var uendeligt lange

For at checke det første tilfælde bruger vi de to vejstykker **p**, som går fra (0 , 1) til (1 , 0) og **q**, som går fra (0 , 0) til (1 , 1).

De krydser hinanden, og vi gennemløber algoritmen med de værdier:

$$\mathbf{a} = \mathbf{q}_2 - \mathbf{q}_1 = (1\,,1) \qquad \mathbf{b} = \mathbf{p}_2 - \mathbf{p}_1 = (1\,,-1) \qquad \mathbf{c} = \mathbf{p}_1 - \mathbf{q}_1 = (0\,,1)$$

$$d = \mathbf{c} \cdot \mathbf{a}^{\perp} = 1 \qquad e = \mathbf{c} \cdot \mathbf{b}^{\perp} = 1 \qquad f = \mathbf{a} \cdot \mathbf{b}^{\perp} = 2$$

Dette betyder at algoritmen ikke returnerer tidligt, da ingen af kravene for at den skulle fejle var opfyldt. Dermed er der krydsning, og denne er i punktet (0.5 , 0.5).

Det næste tilfælde bliver undersøgt med vejstykkerne **p**, som går fra (0 , 0) til (2 , 0) og **q**, som går fra (1 , 1) til (3 , 1). De er parallelle, og krydser dermed aldrig hinanden.

Udregningerne bliver:

$$\mathbf{a} = \mathbf{q}_2 - \mathbf{q}_1 = (2\,,0) \qquad \mathbf{b} = \mathbf{p}_2 - \mathbf{p}_1 = (2\,,0) \qquad \mathbf{c} = \mathbf{p}_1 - \mathbf{q}_1 = (-1\,,-1)$$

$$d = \mathbf{c} \cdot \mathbf{a}^{\perp} = -2 \qquad e = \mathbf{c} \cdot \mathbf{b}^{\perp} = -2 \qquad f = \mathbf{a} \cdot \mathbf{b}^{\perp} = 0$$

Dette fejler, da f ikke er større end 0, samtidig med at d er mindre end f. Dette var forventet, da linjerne er parallelle.

Det sidste tilfælde checkes med **p**, som går fra (0 , 0) til (2 , 0) og **q**, som går fra (1 , 1) til (3 , 1). Disse to linjer ville krydse hinanden hvis de var uendeligt lange (ikke linjestykker, men egentlige linjer).

Udregningerne bliver:

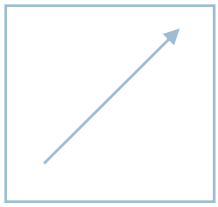
$$\mathbf{a} = \mathbf{q}_2 - \mathbf{q}_1 = (1\,,-1) \qquad \mathbf{b} = \mathbf{p}_2 - \mathbf{p}_1 = (1\,,0) \qquad \mathbf{c} = \mathbf{p}_1 - \mathbf{q}_1 = (-1\,,-1)$$

$$d = \mathbf{c} \cdot \mathbf{a}^{\perp} = -2 \qquad e = \mathbf{c} \cdot \mathbf{b}^{\perp} = -1 \qquad f = \mathbf{a} \cdot \mathbf{b}^{\perp} = -1$$

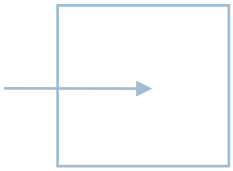
Dette fejler, da f ikke er større end 0, samtidig med at d er mindre end f som før, og var forventet. Linjerne som stykkerne er en del af krydser hinanden, men ikke i det interval hvor stykkerne optræder.

Krydsning mellem et vejstykke og en node i quad-træet

Den brugte algoritme er beskrevet i afsnit 4.5. Vi vil argumentere for algoritmens korrekthed, ved at opstille de forskellige situationer der kan opstå, og checke at den returnerer det korrekte resultat. Algoritmen siger, at hvis Indgang < Udgang, Indgang < 1 og Udgagn > 0, eksisterer der en krydsning. Vi opstiller derfor værdierne for de forskellige tilfælde:

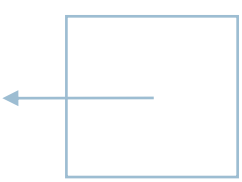


Træ-node: (-1,-1) til (1,1)
Vej: (-0.8,-0.8) til (0.8,0.8)
Indgang = -0.125
Udgang = 1.125
Krydsning = sand

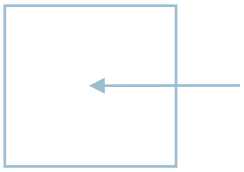


Træ-node: (-1,-1) til (1,1)
Vej: (-2,0) til (0,0)
Indgang = 0.5
Udgang = 1.5
Krydsning = sand

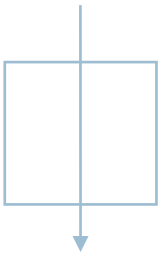
Træ-node: (-1,-1) til (1,1)
Vej: (0,0) til (2,0)
Indgang = -0.5
Udgang = 0.5
Krydsning = sand



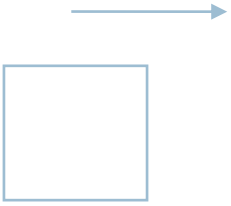
Træ-node: (-1,-1) til (1,1)
Vej: (0,0) til (-2,0)
Indgang = -0.5
Udgang = 0.5
Krydsning = sand



Træ-node: (-1,-1) til (1,1)
Vej: (2,0) til (0,0)
Indgang = 0.5
Udgang = 1.5
Krydsning = sand



Træ-node: (-1,-1) til (1,1)
Vej: (0,2) til (0,-2)
Indgang = 0.25
Udgang = 0.75
Krydsning = sand



Træ-node: (-1,-1) til (1,1)
Vej: (0,2) til (2,2)
Indgang = -0.5
Udgang = -∞
Krydsning = falsk

Det ses at algoritmen giver det korrekte i alle tilfælde.

6.2 Regeltests

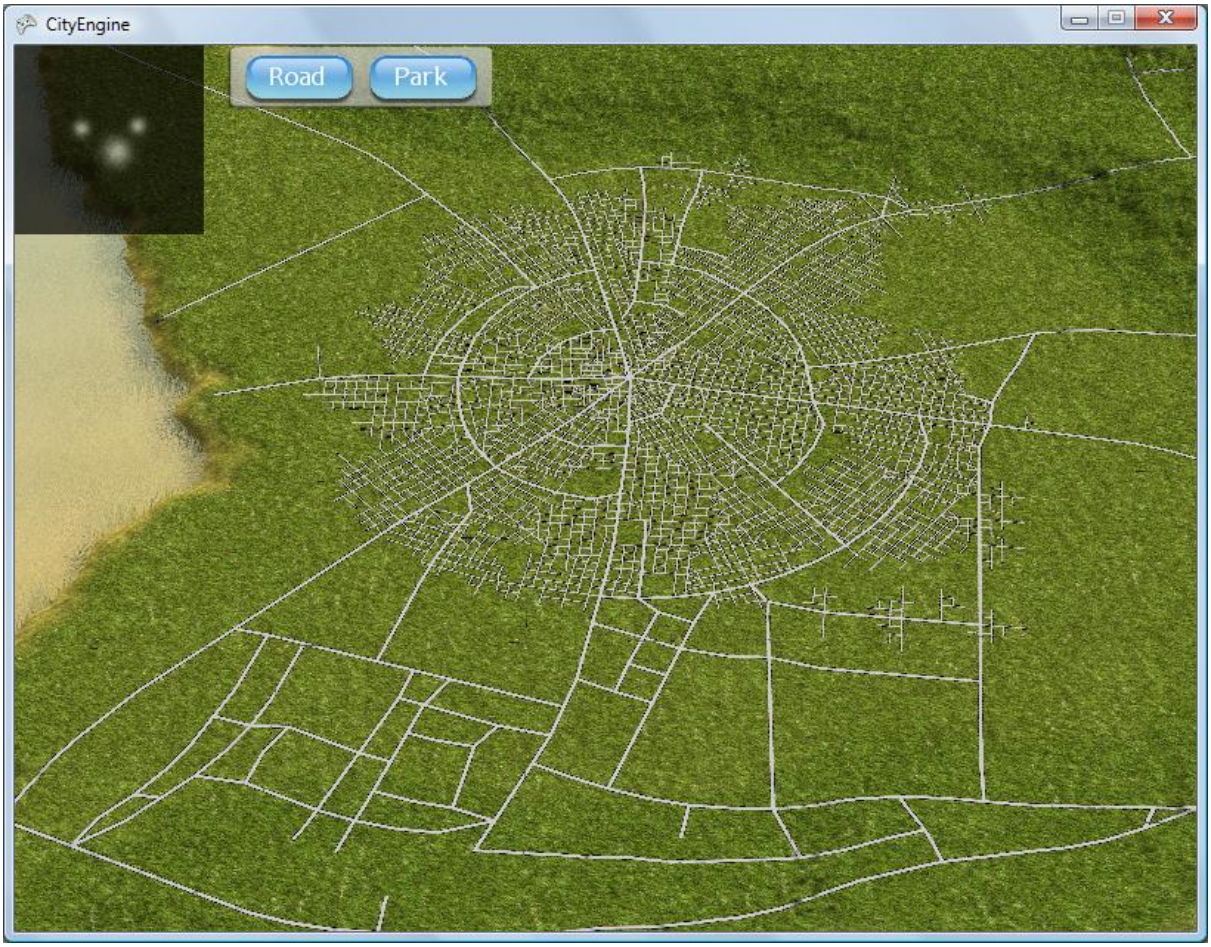
Vi har delt disse illustrationer op i forskellige dele, der omhandler de forskellige regler og deres visuelle indvirken på de genererede byer.

Test af radiale og koncentriske veje omkring et bycentrum

Figur 6-1 Radiale og koncentriske veje omkring et bycentrum

Indsættes et bycentrum vil vejene omkring dette, enten gå radialt ud fra centrum, eller danne ringveje om det. Det ses at den tilfældige påvirkning medfører at ringvejene ikke danner perfekte cirkler.

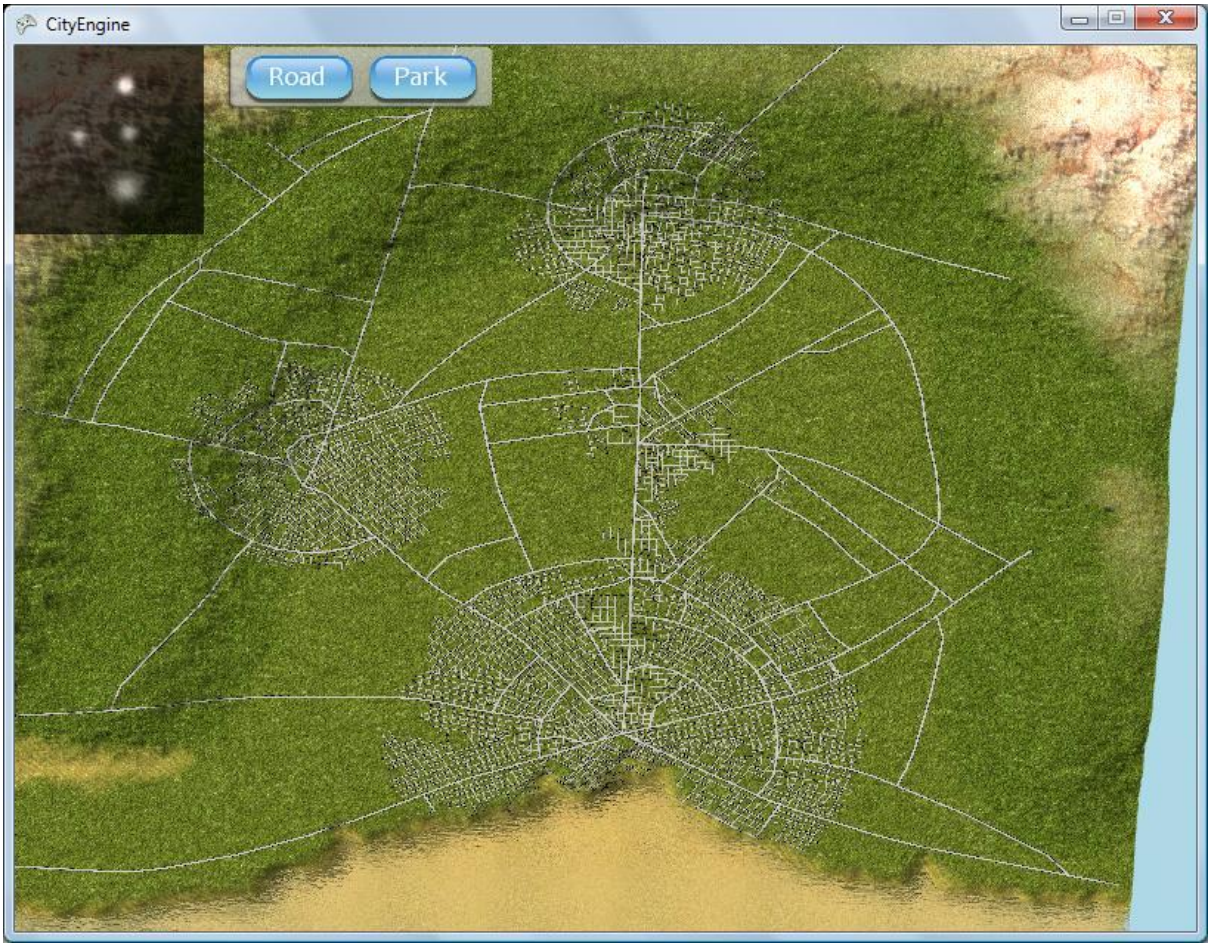
Test af "basic rule" når man kommer væk fra centrum



Figur 6-2 Skifte til "Basic rule" uden for byens centrum

Når vejene når væk fra centrum af byen, dvs. udenfor en ønsket radius omkring det placerede bycentrum, holder de op med at danne ringveje og radiale veje. Som det ses på Figur 6-2, går de over til bare at tage højde for landskabet, og begynder at søge mod det nærmeste bycentrum. Dette er tydeligt når vejene ikke længere går radialt ud, og forgreningerne ikke længere retter sig ind i cirkler.

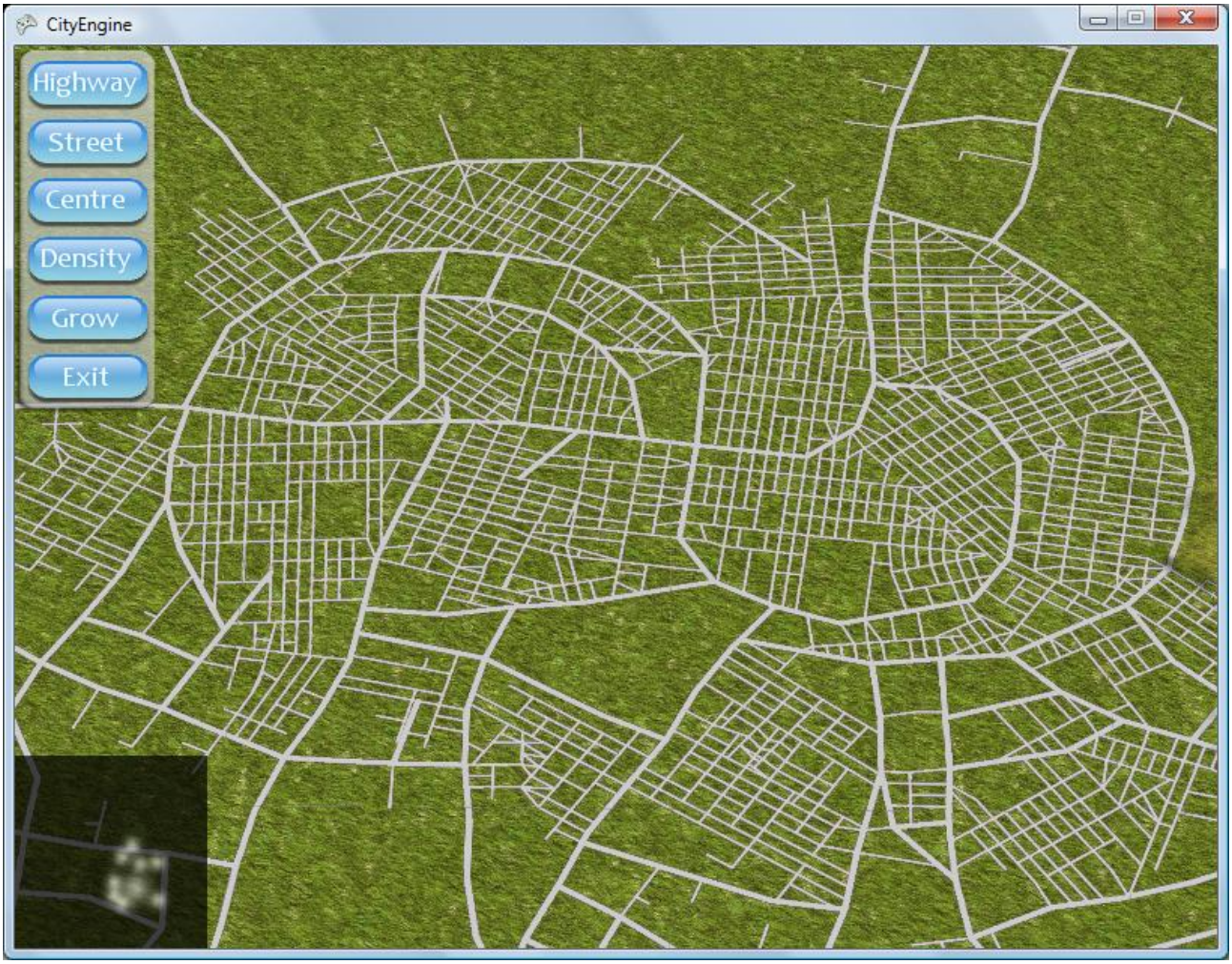
Test af reglerne ved to bycentre der ligger langt fra hinanden



Figur 6-3 Illustration af flere bycentre

Her ses det, at vejene danner ringveje og radiale veje, så længe man er inden for førnævnte radius, skifter så til basic reglen når man kommer væk, for igen at danne ringveje omkring det næste centrum de møder.

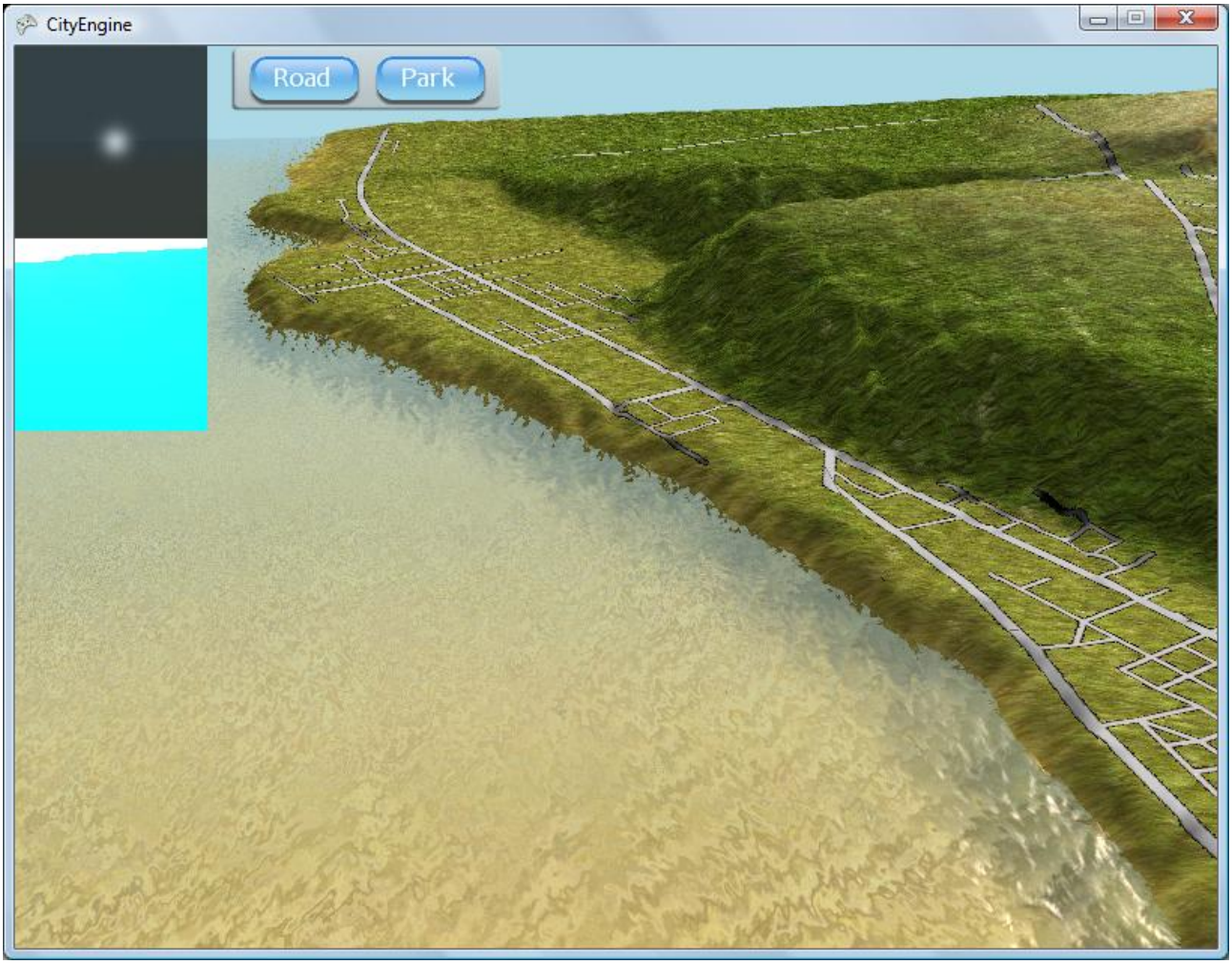
Test af at de koncentriske veje skifter centrum når der er flere bycentre



Figur 6-4 Flere tætliggende bycentre

Vejene danner ringveje omkring det nærmeste bycentrum, og vil derfor skifte hvis der er to, eller flere, centre tæt på hinanden.

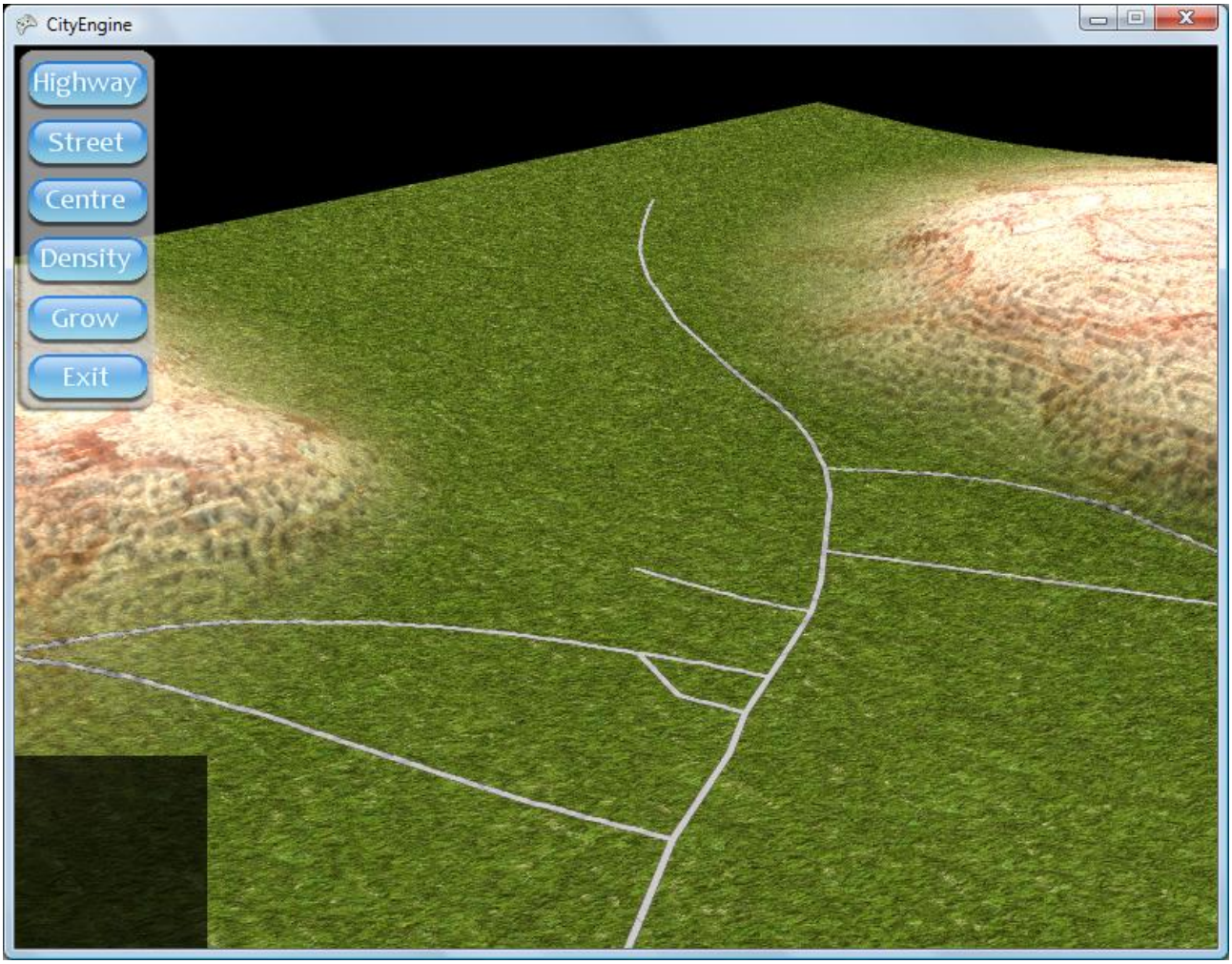
Test af at vejene undviger vandet, og dermed danner kystveje



Figur 6-5 Kystvej

Vejene ser et stykke ud i fremtiden, og vil derfor kunne undgå at ramme vandet, for i stedet for at følge vandkanten væk.

Test af at vejene undviger bjerge



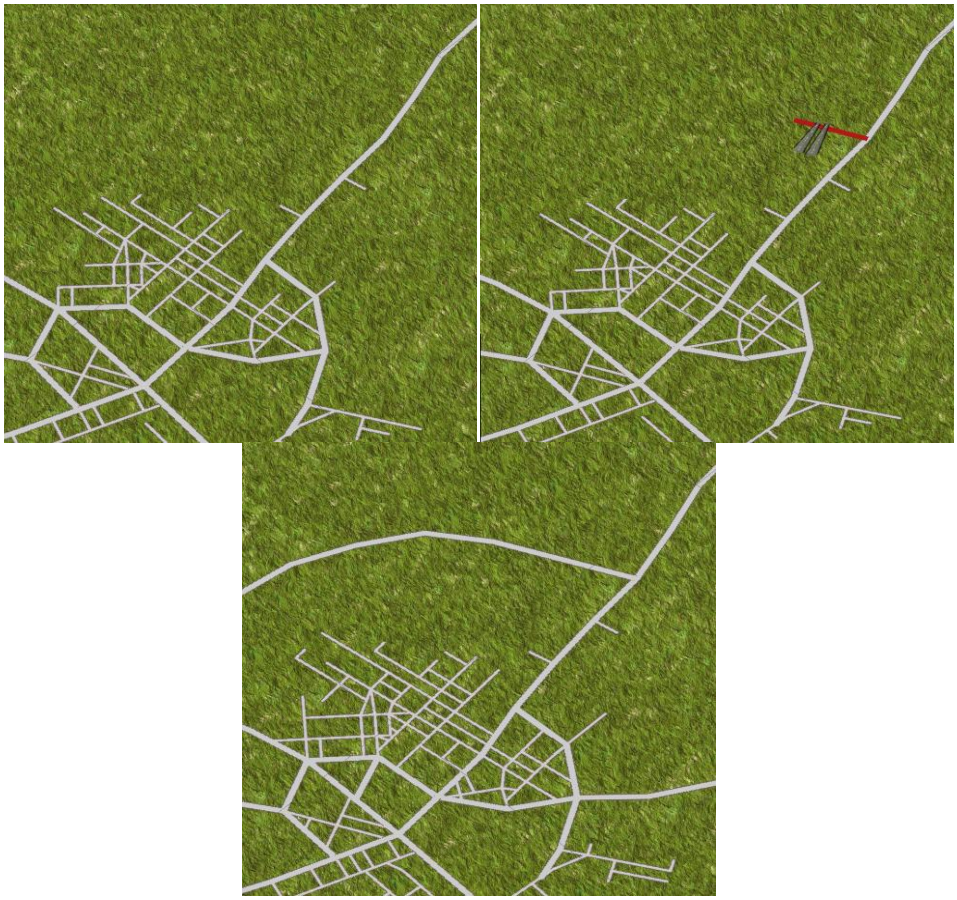
Figur 6-6 Illustration af at vejene finder vej gennem bjerge

Da vejene ser længere frem end bare næste skridt, er det også muligt at finde pas imellem bjerge. De vil altid vælge den laveste stigning, og hvis de kommer i nærheden af bjerge, vil de have en tendens til at undvige dem. Det kan ses ovenfor, hvor vejene bliver afbøjet, når de begynder at nærme sig bjerge, for derefter at følge bjerget rundt. Vejene ser i en vifte foran sig, og hvis de ikke kan finde en position med en rimelig hældning, vokser de ikke videre.

6.3 Manipulationstest

Dette afsnit omhandler muligheden for at influere byen, mens den vokser. Igen er her delt op i forskellige scenarier som illustrerer mulighederne, og hvordan de virker.

Test af indsættelse af hovedveje



Figur 6-7 Illustration af indsættelse af hovedvej

Her ses hvordan en hovedvej indsættes, og at L-systemet tager højde for, og vokser videre fra denne.

Test af indsættelse af gader

Figur 6-8 Udfyldning med gader

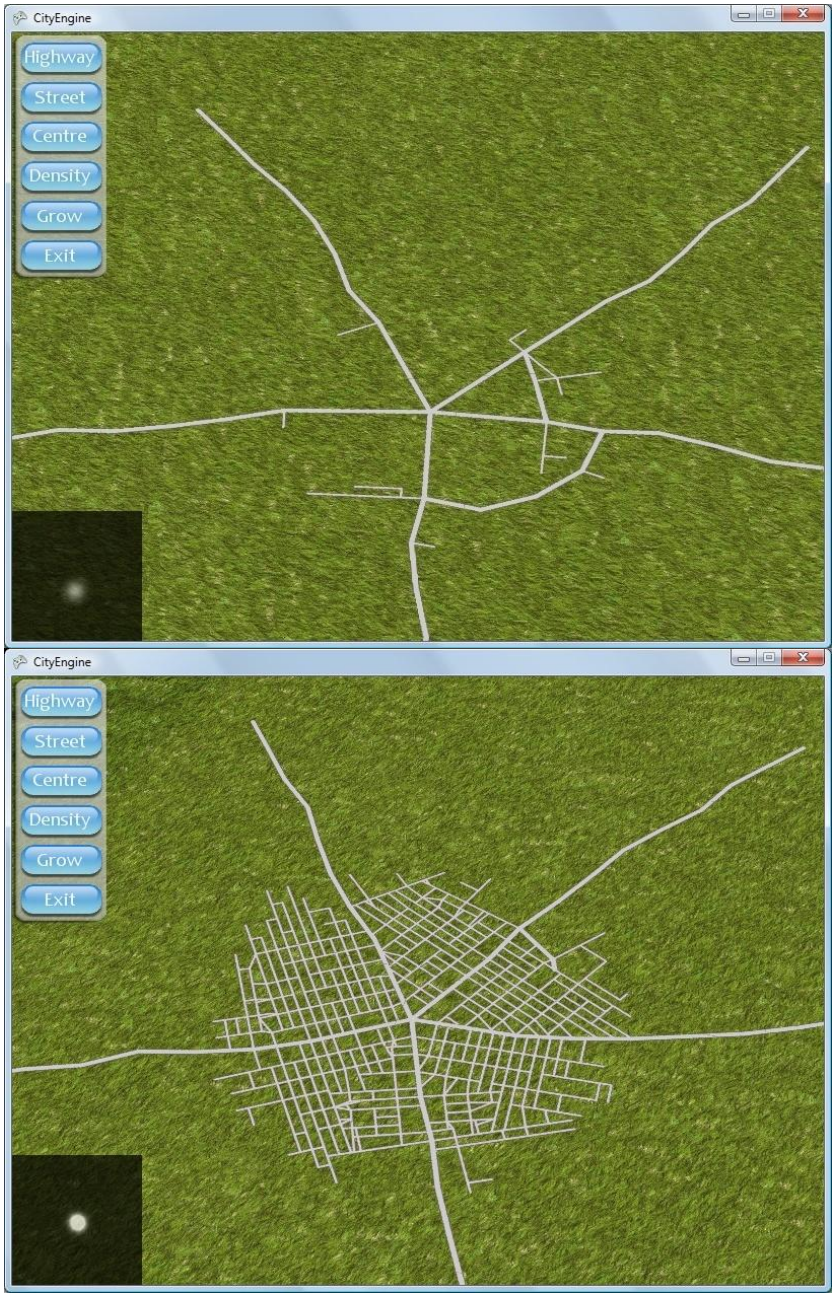
En anden mulighed er at indsætte gader. Dette kan for eksempel bruges, hvis der er et tomt område som ønskes fyldt ud. Igen ses det, at byen tager højde for disse gader, og vokser videre fra, og omkring, dem.

Test af det eksisterende vejnets reaktion på indsættelse af en ny vej

Figur 6-9 Vejnettet finder indsat vej

Det ses, at L-systemet også tager højde for indsatte veje, der ikke er i direkte forbindelse med det eksisterende vejnet. Veje forgrener sig fra det indsatte segment, og andre veje, i nærheden, registrerer det og retter sig ind efter det.

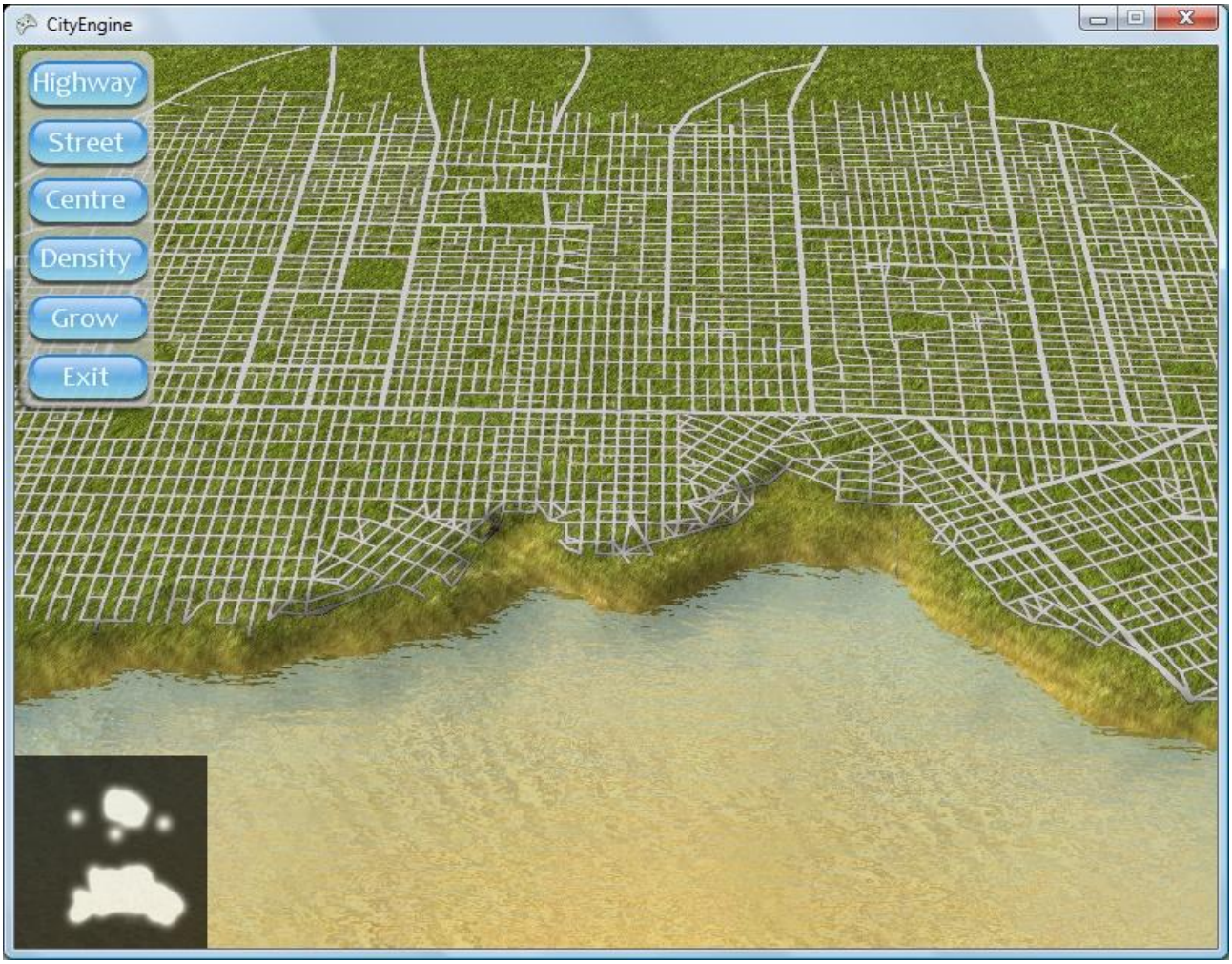
Illustration af befolkningstæthedens "styrke"



Figur 6-10 Hastighedsforskel

Ved at indtegne befolkningstæthed med høj intensitet, vil man opleve, at byen vokser med større hastighed. Dette kan ses på intensitetskortet over befolkningstæthed. Billedet ovenfor viser to byer med samme alder (32 iterationer), men med langt højere befolkningstæthed i den nederste. (Tæthedskortet ses i nederste venstre hjørne).

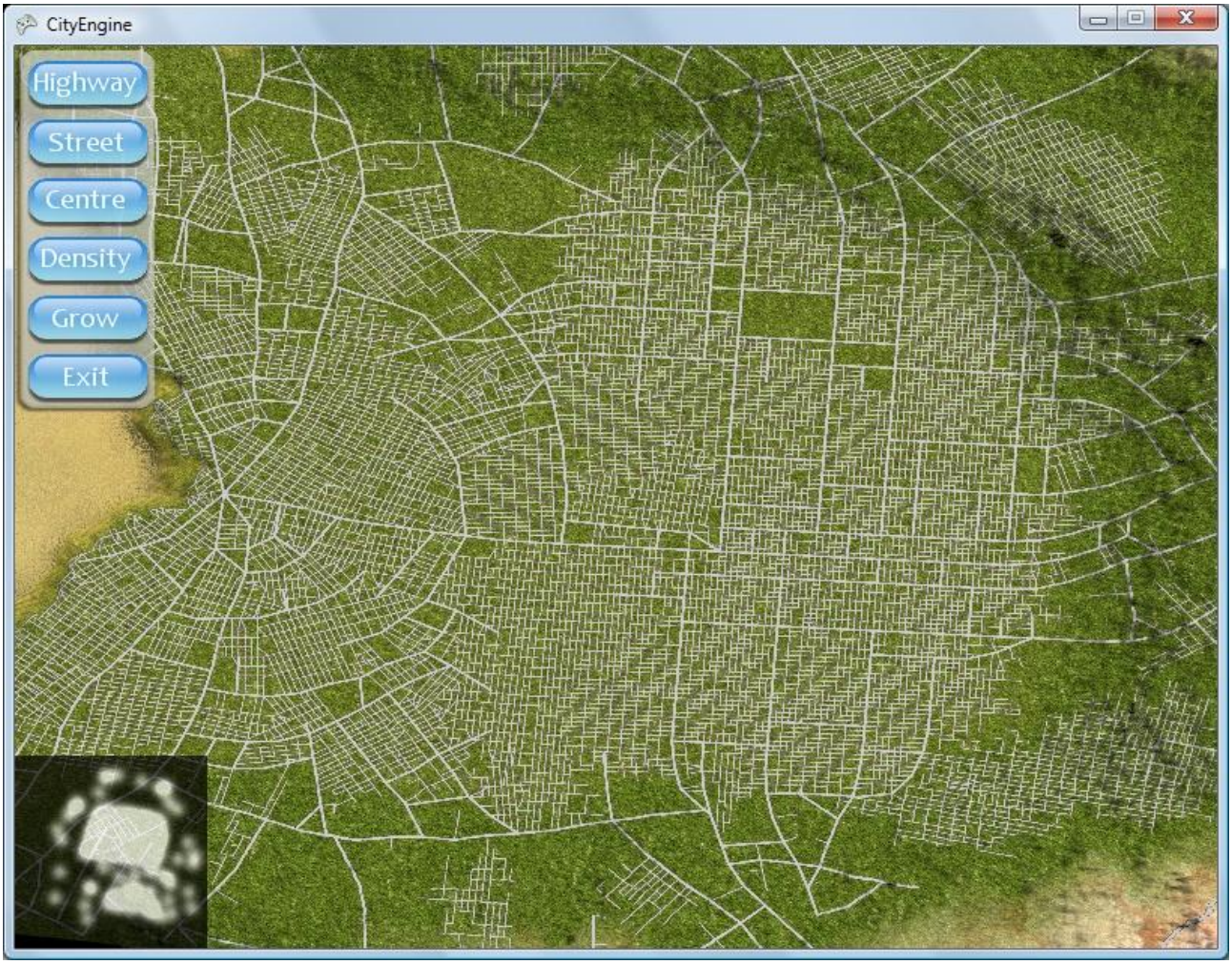
Test af indsættelse af befolkningstæthed uden et bycentrum



Figur 6-11 Befolkningstæthed uden bycentrum

Man kan også vælge at øge befolkningstætheden i et område uden at indsætte et nyt bycentrum. Dette er specielt relevant, hvis man har et eksisterende centrum, og vil øge byens udbredelse. For eksempel et gammelt bycentrum, og et nyere bymiljø omkring dette centrum

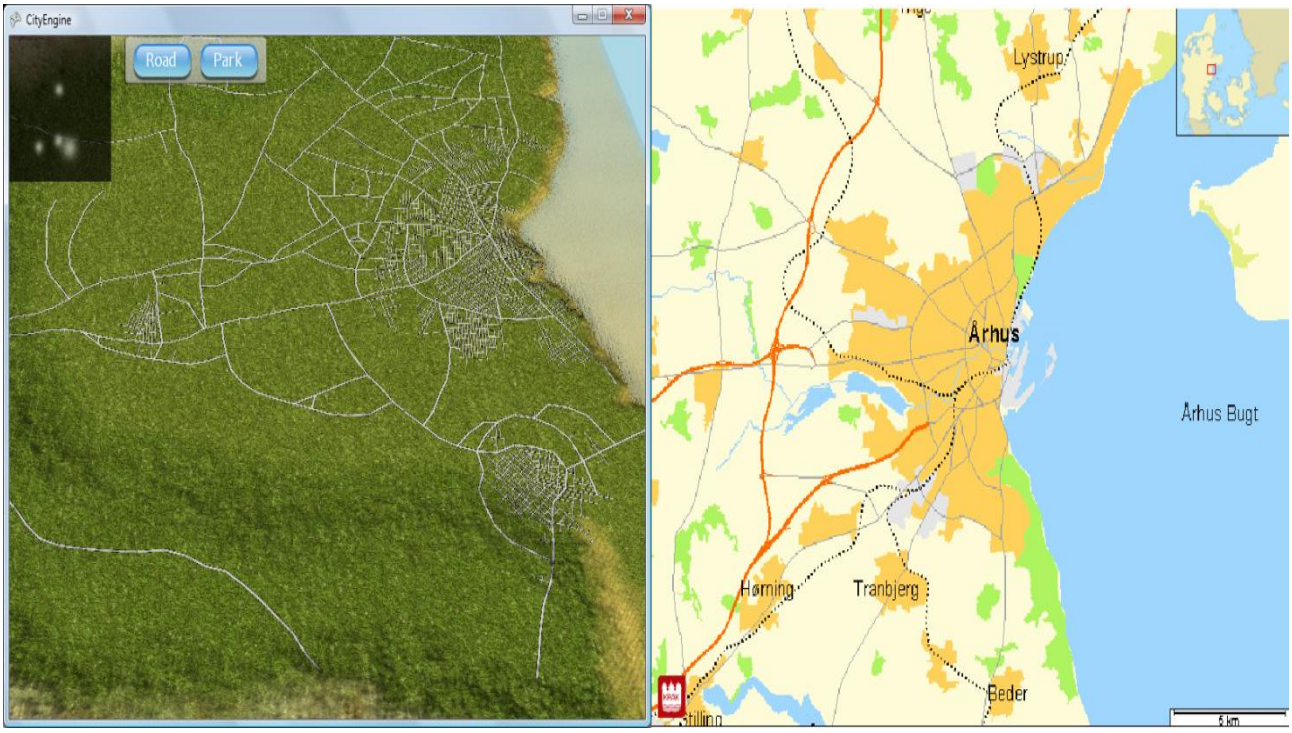
Test af flere bytyper på samme kort



Figur 6-12 Flere forskellige regler påvirker byens vækst

Vi kan specificere flere bytyper ved f.eks. at tegne områder med høj densitet, med eller uden et bycentrum. Billedet overfor viser to byer af forskellig type, der er vokset sammen. Byen til venstre har et centrum og følger derfor den radiale regel, mens byen til højre ikke har et defineret centrum, og derfor følger Manhattan-reglen.

6.4 Realismetest



Figur 6-13 Sammenligning med kort over Aarhus taget fra www.krak.dk

Vi har ønsket, at det skulle være muligt at skabe meget realistiske vejnet. Dette kan bedst testes ved at forsøge at replikere virkelige byer, og se om det er muligt. Det skal helst gøres med så lidt brugerpåvirken som muligt, for at illustrere at det virkelig er programmet, der genererer disse realistiske byer. Idéen med projektet er dog netop, at hvis man ikke er helt tilfreds, kan man selv påvirke væksten.

Figur 6-13 viser, at programmet kan skabe meget realistiske byer. I dette tilfælde en næsten tro kopi af Aarhus. Det vil dog næppe nogensinde lykkes at få proportionerne helt rigtige, da der er utroligt mange faktorer, der spiller ind på virkelige byers vækst. Desuden vil det altid være meget svært at få antallet af veje, og disses størrelse i forhold til hinanden korrekt. Det er svært at bedømme, og endnu sværere at modellere hvor mange gader, der skal være i forhold til hvor mange større veje, og i en virkelig by er der naturligvis mere end blot to forskellige vejstørrelser. Det bliver heller ikke lettere af, at man meget sjældent ser et kort over en by med alle veje tegnet ind.

Disse illustrationer af programmets muligheder dækker den generelle brug af vores udviklede system. De planlagte implementationer er lykkes, og de resulterende byer er interaktive, og realistiske.

7 Diskussion

7.1 Vurdering af resultater

Vurderingen af et system som vores er nødvendigvis subjektiv. Det er svært at formalisere, hvordan en korrekt by tager sig ud. Vi mener at have opnået særdeles realistiske resultater hvad angår det visuelle indtryk af vejnettet. Vi baserer dette på sammenligning med kort over virkelige byer. I vores tilfælde kan vi derudover sammenligne vore resultater med Müllers, da vi har bygget vores metode direkte på hans. Vi vurderer at vores resultater ligger meget tæt på hans.

Vores bestræbelser på at gøre systemet interaktivt må siges at være lykkedes. Vi har bygget et system, der praktisk taget altid reagerer på en brugerens input. Efter vores mening er reaktionerne realistiske og virker naturlige, og man får en fornemmelse af at kunne modellere byen over tid. Man kan styre karakteren af kvarterer, hastigheden hvor med bydele vokser, samt udpege nye områder der ligger ubebygget hen, hvorefter en by vil vokse frem.

Vi har vist hvordan man kan indsætte segmenter i et L-system, hvorfra nye veje kan udvikle sig i de efterfølgende iterationer i L-systemet. Vi mener, at denne funktion bidrager til følelsen af interaktivitet, og en vurdering af resultatet skal således ske ved at betragte systemet i sin helhed.

En vurdering af systemets ydeevne skal ses i forhold til, med hvor stor frekvens L-systemt skal afledes. Det er klart, at et interaktivt system ikke har til formål at modellere en by hurtigst muligt. Vores system skal derimod kunne køre i baggrunden af et spil, og vi mener at dette klart er muligt, selv med en relativt høj udviklingsfrekens.

Resultaterne af modelleringen af træer finder vi særdeles gode. Vi mener, at især sammenføjningerne er blevet vellykkede. Dette skal ses i forhold til hvor simpel løsningen i bund og grund er. Også tegning af blade og løv er efter vores opfattelse ganske realistiske.

7.2 Anvendelse af CityEngine

Vi forestiller os, at et system som vores, hovedsageligt kan finde anvendelse i spil. Som nævnt har vi valgt navnet *CityEngine*, fordi dets opgave er, at håndtere en bys udvikling i en proces for sig, ligesom en fysikmotor varetager et fysisk miljø i et spil.

Systemet giver mulighed for egentlige "levende" byer i spil. Byer der vil udvikle sig med tiden, og som giver en god følelse af en verden der udvikler sig. Har man brug for en statisk kulisseby, vil vi anbefale at man anvender et ikke-interaktivt system som Müllers, da hans metode giver et mere realistisk resultat. Vi har i rapporten forsøgt at redegøre for hvor og hvorfor den interaktive tilgang giver dårligere resultater, og vi mener at artefakterne er relativt begrænsede.

En særlig gruppe spil der kan drage nytte af mere realistiske bymodellerings-systemer er strategispil. Mange spil af denne type er stærkt baserede på at spilleren kan bygge baser, byer eller større samfund. Her kan en motor som vores, være et godt grundlag for computermodstanderen eller en hjælp til spilleren, så man undgår for meget "*micromanagement*"[2].

Der findes en undergenre af strategispillene, der simpelthen kaldes "city-building-games". I denne kategori finder vi spil som *SimCity* og *Caesar*[23]. Vi forestiller os at et system som vores, kan give

[2] Micromanagement er når spilleren skal bruge meget tid på at varetage, og vurdere en masse forskellige parametre og værdier.

brugeren mere generelle kontrolmekanismer. Brugeren bliver fri for at skulle redigere hver eneste lille gade, men har stadig en stor inflydelse på byens udvikling og udseende. Man kan f.eks. lade bygning af skoler, parker, sportanlæg og lignende få indflydelse de kortdata, som L-systemet baserer sig på. Skattesatser og lignende politikker kan influere de globale variable, og derigennem styre byens overordnede vækst. Samtidig har brugeren mulighed for at indsætte veje og styre byens form i detaljer, hvis dette skulle være ønskeligt. Man kunne forestille sig, at brugeren ville have fuld kontrol over særlige områder, så som bymidten.

Generelt ser vi et potentiale i alle spil, hvor der indgår byer på den ene eller anden måde. I et rollespil kunne en bys udseende f.eks. være afhængig af de valg, en spiller tager. I et spil som *Grand Theft Auto* [24], der ofte strækker sig over meget langt tid, ville det være spændende for brugeren med lidt forandring over tid.

7.3 Interaktive L-systemer

Et interaktivt L-system vil være anvendeligt andre steder end ved generering af byer. Forskningen i L-systemer har hidtil været meget fokuseret på at modellere planter og plantedele. Især en gruppe på universitetet i Calgary, ledet af Przemyslaw Prusinkiewicz, arbejder meget med dette ([25]). Vi kunne se en anvendelse, af et system som vores, inden for dette domæne.

I vores *CityEngine* har vi indsat træer, tilfældigt spredt i terrænet. Dette er en meget urealistisk løsning. Det er tidligere vist, at man kan bruge L-systemer til at fordele planter i et landskab [1]. Vi kan bruge metoden med de forsinkede L-systemer i denne proces, og på denne måde skabe en levende skov.

Hastigheden hvormed skove udbreder sig, er afhængigt af de omgivelser den befinder sig i og støder på under sin vækst. Faktorer som adgang til vand og næringsstoffer, forurening, lokale vejrforhold og højde spiller ind. Disse forhold kan ændre sig over tid, f.eks. som følge af menneskelig indgriben. Vi kan modellere dette med vores metode. Ved at lade træernes forsinkelse være afhængige af forholdende, kunne man f.eks. simulere den langsomme udbredelse i sandede næringsfattige områder. Man kunne via et åbent L-system, modellere en gensidig påvirkning af omgivelserne, så skoven over tid ændrede det lokale økosystem til mere gunstige forhold.

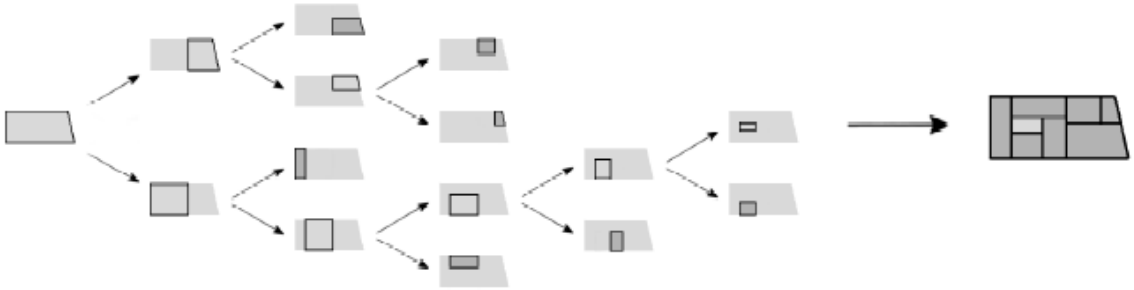
7.4 Fremtidigt arbejde

7.4.1 Indsættelse af huse

Huse er et afgørende element i gengivelse af et bymiljø. Formålet med dette projekt var at gøre den L-system-baserede metode mere interaktiv, og har dermed ikke omfattet huse.

Systemet i sin helhed er dog ikke for alvor brugbart uden huse. Uden huse er vores system sådan set bare en realistisk modellering af vejkort. Vi vil dog argumentere for, at vores tilføjelser til Müllers metode ikke forårsager større problemer for indsættelse af huse.

Müller baserer sin metode på at finde cykler i grafen. En minimalcykel repræsenterer en blok, og den enkelte blok defineres ved et polygon, der udtages af grafen. Dette polygon under-indeles i jordlodder efter metoden skitseret i Figur 7-1. Müller fortsætter herefter med at generere huset, tilpasset de enkelte jordlodder, via endnu et L-system.

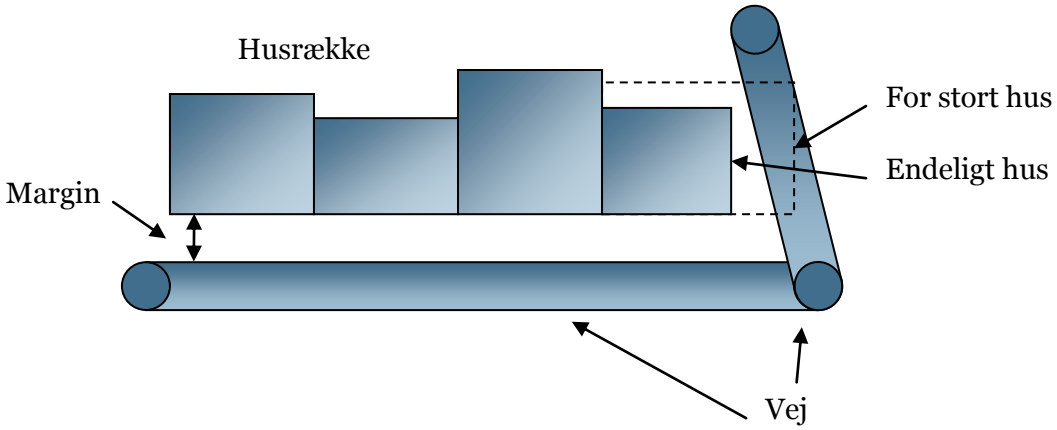


Figur 7-1 Inddeling af jordlodder

Denne metode har vist sig at være rigtigt stærk, og resultaterne er særdeles overbevisende. I vores tilfælde er vi nødt til at generere huse, hver gang vi foretager en afledning af L-systemet. Vi beholder konventionen om, at cyklerne har en begrænset længde. Dette bevirker, at kun små blokke, der normalt kun findes i områder med byudvikling, evaluereres. Vi undgår dermed at vokse huse i områder hvortil byudviklingen ikke er nået.

Der er dog visse problemer med den ovenstående metode. Hvis vi forestiller os, at en blok er bebygget med huse. Vi afleder nu L-systemet, og en ny vej gennemskærer blokken. Vi er med den ovenstående metode, nødt til at beregne hele polygonet på ny, og alle tidligere huse vil blive omdefinerede. Dette er ikke særligt realistisk, og vil formodentligt vække irritation hos en bruger, idet byen ikke er videre konsistent. Metoden giver i øvrigt ikke mulighed for manuel indsættelse af huse, hvilket er afgørernde for fornæmmelsen af interaktivitet.

Vi foreslår en metode som skitseret i Figur 7-2



Figur 7-2 Idé til indsættelse af bygninger

Idéen er at indsætte huse som rækker med tilfældigt valgte intervaller, og med tilfældigt valgte dybder for hvert hus. Rækkens koordinatsystem er defineret udfra vejens normal (Vi skal naturligvis have huse på begge sider af vejen). Når vi indsætter et hus, tester vi for skæring med andre veje og huse. Er huset for stort, forsøger vi med et mindre. Når vi indsætter en vej, kan vi søge efter skæring med de enkelte huse og så kun fjerne disse (ikke hele blokken). Vi kan indføre et modul i L-systemet (i forbindelse med vejmodulet), der gør at husene forsinkes, så de kun opstår nær de centrale dele af byen.

7.4.2 Automatisk afledning af L-system

Skal systemet fungere i spil, er det nødvendigt at byen vokser automatisk, med en specificeret frekvens.

En stor del af processen foregår på CPU'en, og kun renderingen til teksturen er afhængig af GPU'en. Dette er en fordel, da vi på den måde kan reservere GPU'en overvejende til visualisering, mens CPU'en kan udvikle byen. Problemet er at vi, mens vi foretager afledningen af byen, låser brugerfladen. Dette kan let overkommes ved at fortage afledningen i sin egen tråd.

Problemet er nu, at hvis vi forsøger at skrive i L-systemet eller grafen mens den opdateres, vil der opstå inkonsistens i vores data, hvilket kan føre til alvorlige fejl [26]. En simpel løsning på dette problem kunne være at indføre en konvention, der siger, at vi kun kan interagere med systemet ved f.eks. at pause procesen. En mere elegant løsning kunne være at benytte en passende *reader-writer* algorime [26].

7.4.3 Level Of Detail

Et stort problem i vores nuværende system er, at opløsningen af vej-teksturerene er særdeles begrænset, og at vi dermed ikke får en særlig god gengivelse af detaljer som f.eks. fortov og vejstriber.

Dette kunne forbedres ved at benytte metoder til *Level-of-Detail(LoD)*. En simpel metode ville være at markere hver felt i vores terræn med en værdi der indikere hvor langt det er fra kameraet. Og tegne en tekstur, der, i detalje, afhang af denne værdi. Hver gang et felt tegnes vurderes det om teksturen er konsistent med feltets værdi, og denne opdaterers om nødvendigt. Vi vurderer at renderingen af vejene er tilstrækkeligt hurtigt til at understøtte denne metode.

Når huse bliver indsat vil der sansynligvis opstå et behov for *LoD* for disse. Vi mener, at man med fordel kan rendere husene (set fra oven) til samme tekstur som vejene, hvis de er langt fra kameraet. Vi vil på denne måde opnå en effekt der minder om den man ser i flysimulatorer, og byer opfattes på lang afstand som grå flader.

7.4.4 Indsættelse af træer

Indsættelse af træer er i den nuværende implementering meget naiv. En smartere måde at indsætte træer kunne være baseret på L-systemer som skitseret i det foregående. Man kunne på denne måde simulere at træerne vokser i samlede klumper som skove. Det er klart at vi også, i højere grad, kan bruge træerne som inventar i byen, f.eks. sådan at de vokser langs alléer, i parker og på pladser.

For selve *BoP* systemet ville en mulig fremtidig udvidelse være en stærkere parser til brugerdefinerede L-systemer. Vi forestiller os at brugeren skal kunne indtaste L-systemer, med den nøjagtige syntaks der er specificeret i [1], i en tekstfil. Vi vil bruge YACC [27] eller lignede compiler værktøjer til at realisere dette.

8 Konklusion

Vi har implementeret Pascal Müllers metode til beskrivelse af byudvikling i C# og XNA, og udviklet en grafikmotor, der kan vise de genererede byen i et terræn. Vi har derved skabt nogle meget realistiske vejnet, som tager højde for mange parametre som landskab, befolkningstæthed, og ønsket by-udseende.

Derudover har vi videreudviklet Müllers metode, og gjort den mere interaktiv. Med vores model, vokser byerne løbende, og kan på ethvert tidspunkt influeres, og ændres af brugeren. Det er muligt at indikere hvor på landskabet, man vil have nye byer, og hvor hurtigt disse skal vokse frem. Derudover kan man på ethvert tidspunkt indsætte nye veje (både hovedveje, og mindre gader) som L-systemet så vil inkludere, og tage med når det vokser.

Vores system gør det muligt at ”tegne” et indledende bycentrum hvorfra byen kan udvikle sig.

Vi har foretaget en funktionel test af systemet, og på baggrund af denne, tør vi konkludere at systemet generelt opfylder de specificerede krav.

Det implementerede system er beskrevet og dokumenteret i denne rapport.

Appendiks

Kildehenvisninger

[1]. **Prusinkiewicz, Przemyslaw and Lindenmayer, Aristid.** *Algorithmic Beauty of Plants.* New York : Springer-Verlag, 1990.

[2]. **Maxis.** *SimCity.* [Online] http://www.simcity.com.

[3]. Transport Tycoon. *Wikipedia, the free encyclopedia.* [Online] http://en.wikipedia.org/wiki/Transport_Tycoon.

[4]. **Takase, Y., et al.** *Automatic Generation of 3D City Models and Related Applications.* Tokyo, Japan : s.n.

[5]. **Brenner, Claus.** *Towards Fully Automatic Generation of City Models.* Institute for Photogrammetry, Stuttgart University. Amsterdam : s.n., 2000.

[6]. *A Survey of Procedural Techniques for City Generation.* **Kelly, George og McCabe, Hugh.** Dublin, Ireland : s.n.

[7]. *Real-time procedural generation of `pseudo infinite' cities.* **Greuter, Stefan, et al.** s.l. : ACM Press, 2003.

[8]. **Lechner, Thomas, et al.** *Procedural City Modeling.* 2003.

[9]. **Green, Mark, et al.** *Template-based generation of road networks for virtual city modeling.* 2002.

[10]. **Müller, Pascal.** *Prozedurales Modelieren einer Stadt.* Computer Graphics Laboratory, ETH Zürich. 1999. Semester thesis.

[11]. *Procedural Modeling of Cities.* **Müller, Pascal og Parish, Yoav I. H.** [red.] Eugene Fiume. New York : ACM Press, 2001. SIGGRAPH 2001. ISBN: 1-58113-374-X.

[12]. **Interactive Data Visualization, Inc.** SpeedTree | IDV, Inc. *SpeedTree.* [Online] http://www.speedtree.com/.

[13]. *Visual Models of Plants Interacting with Their Environment.* **Mech, Radomir og Prusinkiewicz, Przemyslaw.** Calgary, Alberta, Canada : University of Calgary, 1996.

[14]. **Müller, Pascal.** CityEngine. *Pascal Mueller's Wiki.* [Online] http://www.vision.ee.ethz.ch/~pmueller/wiki/CityEngine/Front.

[15]. —. *Design und Implementation einer Preprocessing Pipeline zur Visualisierung prozedural erzeugter Stadtmodelle.* Computer Graphics Laboratory, ETH Zürich. 2001. MSc Thesis.

[16]. *Procedural Modeling of Buildings.* **Müller, Pascal, et al.** New York : ACM Press, 2006. SIGGRAPH 2006. ISSN: 0730-0301.

[17]. **Introversion Software.** *Introversion Software.* [Online] http://www.introversion.co.uk/.

[18]. It's all in your head, Part 3. *Introversion forum*. [Online] http://forums.introversion.co.uk/introversion/viewtopic.php?t=586.

[19]. **Williams, Lance.** *Casting Curved Shadows on Curved Surfaces*. Computer Graphics Lab, New York Institute of Technology. 1978.

[20]. *Shadows Algorithms for Computers Graphics*. **Crow, Frank.** 1977. SIGGRAPH 1977.

[21]. **Wolfram Research, Inc.** von Neumann Neighborhood -- from Wolfram MathWorld. *Wolfram MathWorld*. [Online] http://mathworld.wolfram.com/vonNeumannNeighborhood.html.

[22]. **Akenine-Möller, Tomas og Haines, Eric.** *Real-Time Rendering, Second Edition*. s.l. : A K Peters, Ltd., 2002.

[23]. *Tilted Mill Entertainment*. [Online] http://www.tiltedmill.com/.

[24]. *Rockstar Games*. [Online] http://www.rockstargames.com/.

[25]. Algorithmic botany - publications. *Algorithmic Botany*. [Online] http://algorithmicbotany.org/papers/.

[26]. **Andrews, Gregory R.** *Foundations of Multithreaded, Parallel, and Distributed Programming*. University of Arizona : Addison Wesley, 2000. ISBN 0-201-35752-6.

[27]. **Johnson, Stephen C.** Yacc: Yet Another Compiler-Compiler. *The Lex & Yacc Page*. [Online] http://dinosaur.compilertools.net/yacc/index.html.

[28]. **Microsoft Corporation.** *XNA Developer Center*. [Online] http://microsoft.com/xna.

[29]. —. XNA Frequently Asked Questions. *msdn*. [Online] http://msdn2.microsoft.com/en-us/directx/aa937793.aspx.

[30]. —. API Migration Guide: Managed DirectX 1.1 to XNA Framework (Beta). *Managed DirectX to XNA Framework Migration Guide*. [Online] http://msdn2.microsoft.com/en-us/xna/aa937797.aspx.

Figurliste

Figur 2-1 Moduler i *Anabaena Cantenula*.....10
Figur 2-2 Visualisering af de 4 første afledninger af L-systemet.....11
Figur 2-3 Skildpaddetolkning af modulerne '+', '-' og 'F' 11
Figur 2-4 De fem første trin af *Koch Island. Figurerne er genereret med Fractals*.....12
Figur 2-5 Tolkning af modulerne '+', '-', '&', '^','\' og '/'.....13
Figur 2-6 Forgrenet L-system.....13
Figur 2-7 Tre forskellige modeller genereret af samme L-system.....15
Figur 2-8 Parametrisk L-system (model genereret i Fractals) 17
Figur 2-9 To grene konkurrerer om lys 19
Figur 2-10 Oversigt over Müllers system.....20
Figur 2-11 Billede fra [11] 22
Figur 2-12 Den mulighed med højst intensitet bliver valgt 23
Figur 2-13 Tre eksempler på regler, der giver meget forskellige vejnet. Billede fra [11] 23
Figur 2-14 Hhv. vejkrydsning, vejkryds undersøgelse, og forlængelse 24
Figur 2-15 Fejl i kanten af byen 24
Figur 3-1 UML-diagram over BoP 25
Figur 3-2 Tegning af et segment 29
Figur 3-3 Segmenter renderet som cylindere. 30
Figur 3-4 Segmenter føjet sammen ved at dele vertices.....30
Figur 3-5 Rotations artefakt.....31
Figur 3-6 Indeksering mellem to segmenter 31
Figur 3-7 Sammenføjning mellem grene 32
Figur 3-8 Fladt segment forårsaget af rotation 32
Figur 3-9 Tropisme. Venstre: Gravitation. Højre: Vind. 33
Figur 3-10 Teksturkoordinater; s = rød, t = grøn 34
Figur 3-11 Sammenligning. Lyset kommer fra samme side på begge billeder 35
Figur 3-12 Belysning af kugle.....35
Figur 3-13 Trækrone der kaster skygge på sig selv 36
Figur 3-14 Trækrone i silhuet 36
Figur 3-15 Diffusemap og normalmap.....37
Figur 3-16 Det endelige resultat.....38
Figur 4-1 Illustraion af pipeline'en i vores system.....39
Figur 4-2 UML-diagram for de tre delprocesser 40
Figur 4-3 UML-diagram over arkitekturen af L-systemet.....41
Figur 4-4 UML-diagram over omgivelserne.....42
Figur 4-5 UML-diagram af grafikmotoren 43
Figur 4-6 De tre ideelle forgreninger.....45
Figur 4-7 Vægt af Manhattanregel som funktion af befolkningstætheden (middelværdi 128).....46
Figur 4-9 Hovedvej forsøger at undgå vand og kuperet terræn 47
Figur 4-8 Beregning af punkthældning 47
Figur 4-10 Vinkel i forhold til centrum.....48
Figur 4-11 Bestemmelse af en vejs retning.....48
Figur 4-12 Radialregel, med ringveje og infaldsveje samt en mindre tilfældigheds faktor.....49
Figur 4-13 Sammenstød mellem to kvarterer.....51

Figur 4-14 Sammenstød mellem to kvarterer, hvis vejene ikke kender de i samme afledning indsatte..... 51
Figur 4-15 Beregning af vinkel til nærmeste eksisterende vejkryds 52
Figur 4-16 Ideel opbygning af blokstruktur 53
Figur 4-17 Opbygning af blokstruktur med varierede forsinkelser 53
Figur 4-18 For stor søgeradius 53
Figur 4-19 Blokstruktur-artefakt 54
Figur 4-20 Søgen efter nærmeste nabo 56
Figur 4-21 Tegning af vejsegmenter og vejkryds 56
Figur 4-23 Et vejsegment renderet med stor forskydning i lag 57
Figur 4-22 Fyldte vejkryds og vejsegmenter 57
Figur 4-24 Visualisering af vejnet 58
Figur 4-25 Prøblem med terræn 58
Figur 4-26 Løsning på førnævnte problem 58
Figur 4-27 Illustration af vores valg 59
Figur 5-1 Fra Pascal Müllers test for Manhattan, visualiseret i 3 trin. 62
Figur 6-1 Radiale og koncentriske veje omkring et bycentrum 68
Figur 6-2 Skifte til ”Basic rule” uden for byens centrum 69
Figur 6-3 Illustration af flere bycentre 70
Figur 6-4 Flere tætliggende bycentre 71
Figur 6-5 Kystvej 72
Figur 6-6 Illustration af at vejene finder vej gennem bjerge 73
Figur 6-7 Illustration af indsættelse af hovedvej 74
Figur 6-8 Udfyldning med gader 74
Figur 6-9 Vejnettet finder indsat vej 75
Figur 6-10 Hastighedsforskel 76
Figur 6-11 Befolkningstæthed uden bycentrum 77
Figur 6-12 Flere forskellige regler påvirker byens vækst 78
Figur 6-13 Sammenligning med kort over Aarhus taget fra www.krak.dk 79
Figur 7-1 Inddeling af jordlodder 82
Figur 7-2 Idé til indsættelse af bygninger 82

Introduktion til XNA

XNA [28], som står for ”**X**NA’s **N**ot **A**cronymed”[29], er Microsofts nyeste version af Managed DirectX(MDX). MDX 2.0 var under udvikling, men arbejdet blev afbrudt til fordel for XNA. Version 1.0 udkom i december 2006, og fik i april 2007 en opdatering.

Idéen med XNA er at det skal være nemmere, og mere let tilgængeligt, for hobby spilprogrammører og lignende at lave spil. Microsoft vil gøre det lettere for ”almindelige mennesker” at lave små spil, og dele dem med andre ligesindede. De vil også gerne have folk, der er under uddannelse og mindre, uafhængige spilfirmaer (independent game developers) til at bruge deres teknologi. De prøver at gøre det let og hurtigt at komme i gang ved at gøre meget af ”benarbejdet” for programmøren. I XNA er det meget hurtigt og let at importere og bruge billeder som teksturer, importere modeller i forskellige formater. Desuden er brugen af shader-effektfiler gjort lettilgængelig. Derudover er der klasser der repræsenterer det meste af, hvad man får brug for til geometriske beregninger. Datastrukturer som kugler, flader, rays og bounding boxes er alle implementeret med tilhørende skæringsalgoritmer og lignende. Dette gør det let for begyndere, der vil i gang med at lave spil, og lader dem komme i gang hurtigt. XNA kodes i *XNA Game Studio Express* (GSE), og er disse er indkorporeret fuldstændigt. Dette giver fejlfinding og anden *debugging* lige så hurtig som det er i C# generelt (i Visual Studio).

XNA bygger på DirectX 9.0c, og har, med få undtagelser, de samme muligheder som eksisterer i dette. Desuden har XNA en stor del af sit interface tilfælles med MDX, og programmører herfra har let ved at migrere. En *migration guide* gives i [30].

XNA lider ligesom MDX af performance problemer, da det skal afvikles på en virtuel maskine. Mange flaskehalse, f.eks. i forbindelse med resourcetilgang, er dog efterhånden minimeret i en grad, så man i de fleste tilfælde ikke mærker nævneværdig forskel på XNA og directX (med mindre man virkeligt har brug for alle maskinens kræfter). Hastigheden af programmering og udvikling, samt især fejlfinding er til gengæld øget så meget, at det ofte vil være mere attraktivt, på trods af tabet i ydelse. Dermed er det netop også interessant for ”almindelige mennesker” der hellere vil bruge deres tid på at lave nye spændende features, end at fejlfinde, og rette deres C++ kode.

Microsoft har udviklet XNA, så det kan bruges både på en Windows PC, og på deres spillekonsol, Xbox 360. Et program/spil skrevet i C# og XNA kan kompileres direkte på en PC og en Xbox hvis man følger ganske få regler. Dette giver mange muligheder, og via deres online samfund ”Microsoft Live” lader det folk dele de spil, de har lavet med andre lige sindede.

Problemet ved denne Xbox kompatibilitet er, at man er hardwarebegrænset. Nu er DirectX 10 netop kommet, og hvor en Windows-bruger kan opgradere sit grafikkort og derved få adgang til alle de nye muligheder, er en Xbox bruger begrænset. Dette medfører, at DirectX 10 features ikke umiddelbart bliver tilgængelige gennem XNA. Fordelen ved konsol kompatibiliteten er til gengæld, at man ikke skal tænke over hvilken slags hardware slutbrugeren sidder med. Har han/hun et grafikkort, der understøtter shadermodel 2.0, kan et program, udviklet i XNA, afvikles.

En af de helt store nyskabelser er adgangen til en *content pipeline*, og den måde hvorpå XNA i det hele taget håndterer indhold. En almindelig programmør skal praktisk taget ikke tænke på hvordan modeller, teksturer og effektprogrammer hentes ind i programmet. Man tilføjer simpelthen resourcerne i sit GSE-projekt, og XNA vil sørge for at compile disse til et specielt binært

resourceformat (.xnb), der hentes når spillet loades. Denne fremgangsmåde bevirker, at man undgår meget lange indlæsningstider.

XNA understøtter som udgangspunkt en lang række populære formater (f.eks. .x og .fbx). Har man brug for andre typer, kan man enten selv programmere en *custom-content-pipeline* eller hente brugergenererede.

Kendte fejl

Vi har kendskab til følgende fejl i systemet:

Hovedveje stoppes når de rammer en gade, og vi har dermed en stor del "blinde" hovedveje, hvilket således ikke giver indtryk af et net af hovedveje. Vi har valgt denne løsning i mangel på bedre. Det er ikke realistisk at lade vejene fortsætte, da dette har tendens til at forårsage en overflod af veje omkring eksisterende byområder. Løsningen består i at tilpasse værdierne for forgrening af hovedveje, samtidig med at de får lov til at vokse videre, men ikke forgrene sig.

Når fejlbehæftede L-systemer importeres i *BoP*, giver systemet generelt ubrugelige fejlmeddelelser. Vi planlægger en total omskrivning af denne del af systemet, som beskrevet i afsnit (.....).

Strukturen af en by i en meget tidelig fase, er præget af mangel på gader. Vi vurdere at dette i nogen grad kan afhjælpes ved at tilpasse de globale variable. Fejlen kan dog i store træk tilskrives Müllers metode, og opvejes af det faktum at brugeren selv kan indtegne den tidlige by.