Índex

Capítol 1: Introducció

Avui en dia el modelatge de ciutats és un problema obert pel que no existeixen solucions estàndards ni de codi lliure. Aquest és un problema força important per a indústries com la del cinema, la realitat virtual o els videojocs. Dins d'aquest problema es poden presentar sub-problemes interessants, dels quals el modelatge de les xarxes de carrers pren un rol fonamental. La complexitat geomètrica que té una ciutat fa que el modelatge d'aquest tipus d'estructures sigui una tasca gens menyspreable.

Qualsevol persona, amb més o menys relació amb el món dels ordinadors, haurà jugat a algun joc d'aventures gràfiques on l'escenari escollit és una gran ciutat, o haurà vist alguna pel·lícula relativament actual on el protagonista viu la seva aventura envoltat de grans edificis. Cada vegada més es fan servir eines informàtiques per a crear aquests escenaris monumentals. Molts d'aquests escenaris han estat creats manualment, és a dir, muntant els edificis un per un fins aconseguir una ciutat.

El modelatge procedural pretén solucionar bona part d'aquest problema utilitzant les seves funcionalitats per a generar de forma sistemàtica la geometria necessària en cada cas. A grans trets, el procediment consisteix en deixar per la màquina la feina de crear la ciutat pròpiament dita i així l'usuari només s'ha de preocupar d'introduir els paràmetres necessaris per aconseguir el seu propòsit.

Existeixen eines de desenvolupament procedural, però el seu ús es troba restringit a unes quantes aplicacions que, generalment, no inclouen xarxes de carrers.

1.1 Objectius

L'objectiu d'aquest projecte és el desenvolupament d'una eina pel modelatge procedural de ciutats i xarxes de carrers. El modelatge de carrers és, per si sol, un bon tema on aplicar-hi la programació procedural. Les ciutats solen comptar amb patrons que es van repetint al llarg del territori. El fet de "repetir" una tasca suggereix sempre l'aplicació d'algun tipus de procediment per tal de simplificar i reduir la feina de l'usuari a l'hora de desenvolupar aquesta tasca.

L'objectiu principal d'aquest projecte es centra en el desenvolupament d'una eina de fàcil ús que permeti a l'usuari obtenir de manera ràpida l'estructura bàsica d'una ciutat.

1.2 Abast

Aquest projecte s'ha desenvolupat sobre Houdini, una plataforma genèrica pel modelatge procedural d'objectes. Per aconseguir això s'ha utilitzat com a base la feina descrita als articles "Procedural Modeling of Cities"(Müller & Parish, Procedural Modeling of Cities, 2001) i "Template-Based Generation of Road Networks for Virtual City Modeling" (Sun, Baciu, Yu, & Green, 2002). Es pot organitzar la feina seguint una estructura modular. El primer pas és la definició d'un sistema de derivació per a generar ciutats: a partir d'una estructura de regles, aquestes s'apliquen iterativament per aconseguir una estructura donada.

Això implica:

1. Aplicació d'un conjunt de regles que generi iterativament la xarxa de carrers.
2. Aplicacions d'aquest sistema per generar ciutats a partir del mapa d'una ciutat real.
3. Aplicacions per generar ciutats sintètiques amb mapes de districtes, ús de la terra i densitat de població com a entrades.

1.3 Temporalització

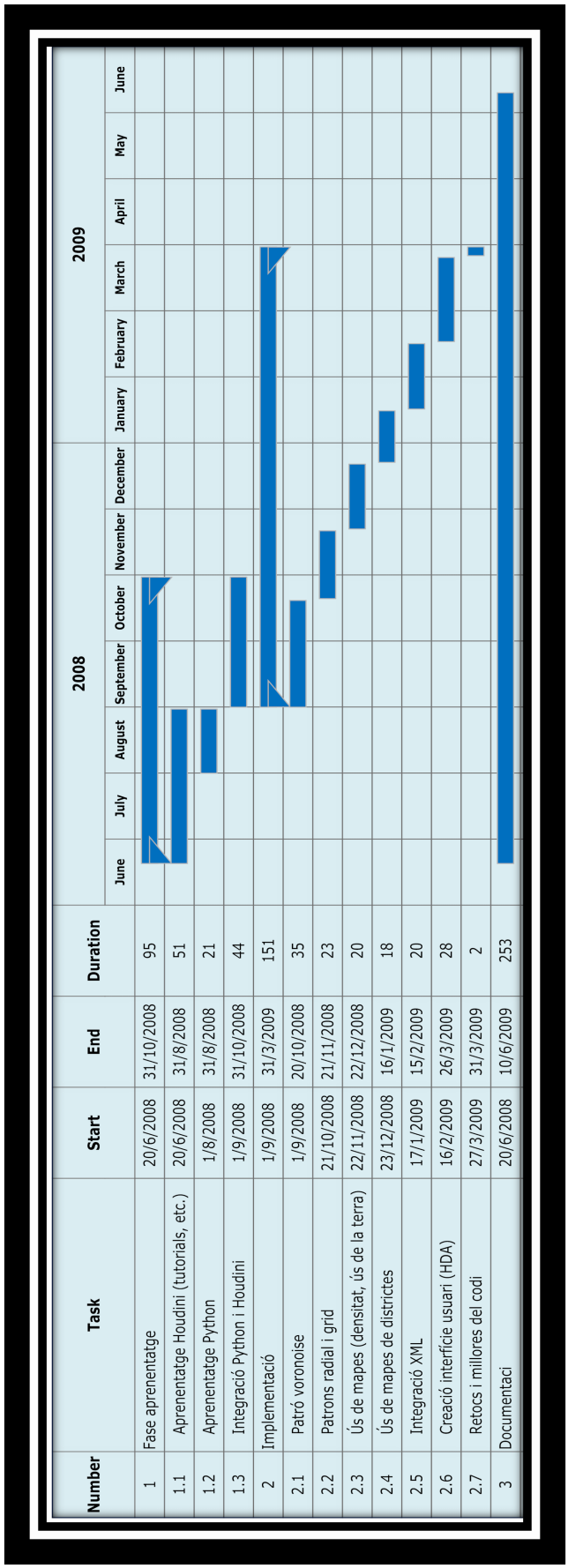
Aquest projecte es va iniciar a principis mitjans de juny de 2008 i s'ha acabat al juny de 2009.

Els primers mesos van estar dedicat a la investigació del funcionament del Houdini i la seva integració amb el llenguatge Python. A partir del setembre es va començar a escriure el codi pel primer patró de ciutat (voronoise), mentre s'anava definint l'abast del projecte. Al novembre, desembre i gener es van implementar la resta de patrons i l'ús de diferents mapes com a entrades.

Durant els mesos de febrer i març es va integrar el codi amb la lectura de paràmetres de fitxers xml i es va crear un HDA que servirà d'interfície d'usuari.

Finalment, durant l'últim mes es va acaber de redactar la memòria.

La Figura 1 mostra un diagrama de *Gantt* amb la temporalització del projecte.



Number	Task	Start	End	Duration
1	Fase aprenentatge	20/6/2008	31/10/2008	95
1.1	Aprenentatge Houdini (tutorials, etc.)	20/6/2008	31/8/2008	51
1.2	Aprenentatge Python	1/8/2008	31/8/2008	21
1.3	Integració Python i Houdini	1/9/2008	31/10/2008	44
2	Implementació	1/9/2008	31/3/2009	151
2.1	Patró voronoise	1/9/2008	20/10/2008	35
2.2	Patrons radial i grid	21/10/2008	21/11/2008	23
2.3	Ús de mapes (densitat, ús de la terra)	22/11/2008	22/12/2008	20
2.4	Ús de mapes de districtes	23/12/2008	16/1/2009	18
2.5	Integració XML	17/1/2009	15/2/2009	20
2.6	Creació interfície usuari (HDA)	16/2/2009	26/3/2009	28
2.7	Retocs i millores del codi	27/3/2009	31/3/2009	2
3	Documentaci	20/6/2008	10/6/2009	253

Figura 1: Diagrama de Gantt

1.4 Estructura del document

Aquesta memòria està estructurada en cinc capítols fonamentals que recullen tota la documentació i explicació necessària del projecte.

El primer capítol conté la introducció al tema central del projecte així com els objectius que es van marcar al inici del mateix i la temporalització estructurada.

Al segon capítol s'expliquen detalladament les eines escollides: Houdini, Python i XML. Es mostren les funcionalitats de cadascuna d'elles i la seva interacció amb la resta.

El tercer capítol mostra els antecedents en què s'ha basat el projecte i es parla dels patrons reals de ciutats en les que s'han basat els patrons desenvolupats en aquest projecte.

El quart capítol consta de l'anàlisi del sistema, amb diagrames de casos d'ús, diagrames d'activitat, diagrama de classes i diagrames de seqüència.

El capítol 5 conté la documentació referent al disseny del projecte, més concretament el disseny del XMLShape i l'ús de les seves regles.

El sisé capítol és complementari al quart, ja que s'hi mostra tota la implementació que s'ha definit en aquest. S'especifiquen tots els conceptes explicats anteriorment i es mostra com s'ha creat el codi del projecte.

El seté capítol recull els resultats obtinguts amb exemples de ciutats construides amb el programa.

El vuité capítol mostra les conclusions extretes dels resultats obtinguts.

Finalment, el capítol 9 proposa el treball futur que es pot desenvolupar a partir del projecte actual.

Al llarg de tot aquest document, quan parlem d'un tipus de node de Houdini, l'escriurem amb majúscules, mentre que quan escriguem el nom d'una variable, serà en minúscula i cursiva.

Capítol 2: Eines

En l'elaboració d'aquest projecte s'han utilitzat bàsicament tres eines diferents: Un modelador 3D, un llenguatge d'scripting i un llenguatge flexible per a definir les ciutats.

Inicialment, es va proposar fer servir només el Houdini com a eina de treball per a modelar la geometria. Tot i això, aquest programa sol no donava la possibilitat de programar l'eina de manera sistemàtica, ja que està orientat a donar servei a través de la interfície gràfica d'usuari.

Per tal de poder implementar un mòdul recuperable i editable externament, es va optar per fer servir el Python que ja està incorporat dins el mateix Houdini. D'aquesta manera obtenim una perfecta connexió entre les dues eines. Primer es crea un script que construeixi la geometria, ja sigui directament amb la consola o usant un editor extern, i automàticament podem visualitzar i editar el resultat dins de Houdini.

Finalment, s'ha optat per dotar al sistema d'una descripció procedural de modelatge a realitzar en el llenguatge XML. Aquesta tecnologia permet a l'usuari treballar amb una sintaxi coneguda i relativament senzilla.

En la secció 2.1 s'explicaran els conceptes bàsics del Houdini i les seves xarxes de nodes. Les xarxes VEX, COP2 i SOP es tractaran més a fons en les seccions 2.2, 2.3 i 2.4 respectivament. La secció 2.5 tractarà sobre la creació de Houdini Digital Assets. En els punts 2.6, 2.7 i 2.8 es parlarà del Python i la seva integració amb el Houdini. Finalment, en el punt 2.9 es parlarà del XML com a llenguatge per al disseny de les xarxes de carrers.

2.1 Modelador 3D: Houdini

El primer pas a fer era escollir un modelador 3D adient i preparat per a suportar el tipus de treball que s'havia planificat. Existeixen diferents programes pensats per a desenvolupar aquest tipus de tasca, però sempre amb lleugeres diferències entre ells. Maya, 3D Studio MAX o Blender són algunes de les opcions que s'havien estudiat.

Maya és un dels softwares més potents del mercat. Destaca sobretot per la seves possibilitats d'expansió i personalització. Utilitza un llenguatge propi anomenat *MEL* que es fa servir per a crear scripts i executar-los sobre el programa.

El 3D Studio MAX és una altra opció dins el món de l'animació 3D. Pertany a la mateixa empresa que Maya però, a diferència d'aquest, 3DStudio és només per Windows. Els dos programes formen part de la mateixa empresa degut a les continues fusions i adquisicions d'empreses més petites propietàries d'aquest software.

Blender és l'opció de software lliure més potent que hi ha al mercat. El programa no té res a envejar als seus germans de pagament, tot i que no disposa de tantes extensions i suplements com ells.

Finalment, i un cop valorat el funcionament dels programes i la feina a desenvolupar es va optar per utilitzar un programa d'animació 3D anomenat Houdini, ja que està especialment pensat per a crear sistemes de modelatge procedural [5].

A diferència dels altres programes esmentats abans, Houdini treballa usant un sistema de nodes en forma de graf acíclic per a representar els objectes de l'escena. D'aquesta manera s'obté un sistema d'objectes independents, units entre ells per un flux de dades.

La Figura 2 mostra l'entorn de treball del Houdini. Aquest entorn està compost per una barra d'eines (superior), des d'on es generen totes les figures i accions. La zona de treball és l'espai on es poden visualitzar els resultats usant diferents vistes i es pot seleccionar i modificar objectes. L'arbre de nodes és l'espai on es representen tots els objectes de l'escena en forma de caixetes independents enllaçades les unes a les altres. Finalment, un cop seleccionat un node, podem consultar i modificar les seves propietats a través del sistema de pestanyes que ens facilita la interfície.

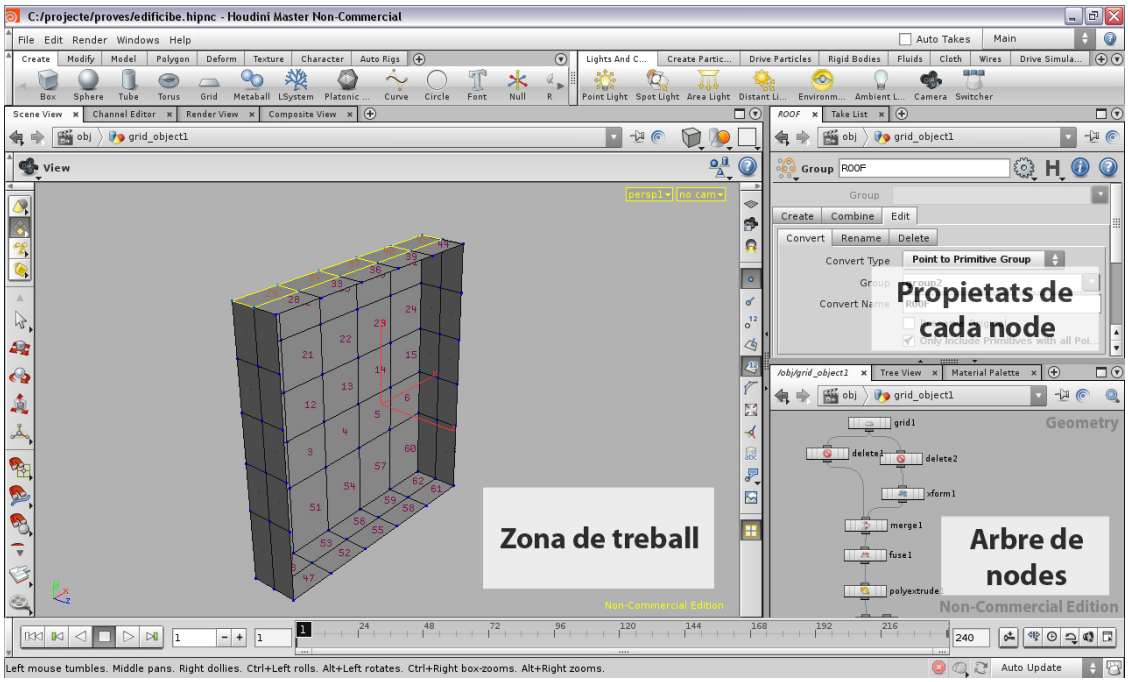


Figura 2: Captura de l'entorn de treball del Houdini.

El Houdini divideix les xarxes de nodes en diferents categories:

- VEX Builder (VOP) – Xarxes per al llenguatge del Houdini anomenat VEX.
- Objects (OBJ) – Xarxes d'objectes (geometria, llum, càmera, etc).
- Geometry (SOP) – Xarxes per construir geometries.
- Particle (POP) – Xarxes per efectes de partícules.
- Compositing (COP2) – Xarxes per composició 2D.
- Dynamic (DOP) – Xarxes per a efectes dinàmics.
- Channel (CHOP) – Xarxes per a operacions de canals. Els canals controlen com canvien els valors a través del temps.
- Shaders (SHOP) – Xarxes per a crear diferents *shaders*.
- Output (ROP) – Xarxes per especificar les opcions de renderització.

Per a aquest projecte només s'han fet servir VOP, COP2 i SOP (que està inclòs en OBJ), les característiques dels quals es comentaran a continuació.

2.2 Shaders de superfície (VOP)

Si volem crear una imatge proceduralment, amb altres software caldria escriure un *shader*[1]. De manera similar, el Houdini compta amb un llenguatge anomentat VEX, que permet crear *shaders.* L'exemple de la Figura 3 crea dos nodes: un del tipus CONSTANT (que explicarem en el punt 2.2.3), el valor del qual serà un color i un del tipus OUTPUT (que explicarem en el punt 2.2.2) i els connecta. Això pintarà la sortida del color definit al node CONSTANT.

```
#define VOP_SHADING
#define VOP_SURFACE
#pragma opname surface1
#pragma oplabel "VEX Builder Surface"
#pragma opmininputs 0
#pragma opmaxinputs 0

surface
surface1(){
    vector      constant;
    constant = { 0.9, 0, 0.9 };
    vector tempCf = constant;
    Cf = tempCf;
}
```

Figura 3: Exemple de codi VEX, que pinta la sortida del color definit

[1] Un shader és un conjunt d'instruccions gràfiques destinades per al accelerador gràfic, que donen l'aspecte final d'un objecte. Els shaders determinen materials, efectes, color, etc.

De totes maneres, no cal fer servir el llenguatge explícitament. Amb el *VEX Builder* (Figura 4) podem crear el codi VEX anterior connectant nodes.

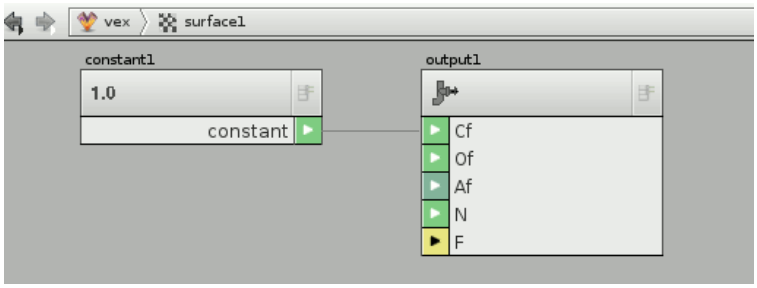


Figura 4: Exemple de construcció d'un shader amb VEX Builder

Per visualitzar els resultats d'aquest contexte, cal obrir una pestanya a la zona de treball del tipus Shader View.

A continuació, explicarem alguns nodes que farem servir en el projecte.

2.2.1 El node GLOBAL

Aquest operador moltes vegades serà el primer node a crear per a una xarxa VEX. Les seves sortides representen totes les variables globals de la xarxa VEX. Nosaltres només farem servir les sortides X i Y, que representen les coordenades de cada punt del pla, com veiem a la Figura 5.

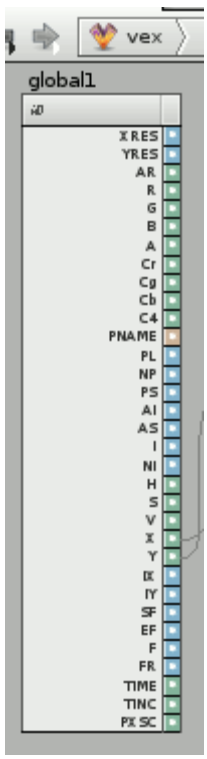


Figura 5: Exemple d'un node GLOBAL.

2.2.2 El node OUTPUT

Tota xarxa VEX necessita un node OUTPUT. Les entrades d'aquest representen les variables globals de resultats de la xarxa actual que poden ser modificades. Les variables que pot fer servir aquest node seran un subconjunt de les variables disponibles pel node GLOBAL. Nosaltres farem servir les entrades R, G i B, que representen els canals de sortida vermell, verd i blau respectivament, com veiem a la Figura 6.

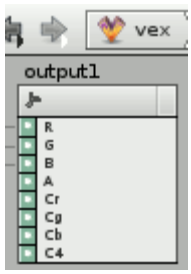


Figura 6: Exemple de node OUTPUT

En l'exemple de la Figura 7 tenim un node CONSTANT (explicat a la secció 2.2.3) que conté un valor constant de tipus color, concretament blau. L'endollem al node OUTPUT a través d'un node VECTOFLOAT, que ens divideix els tres valors del vector que representa el color blau en tres valors de tipus float.

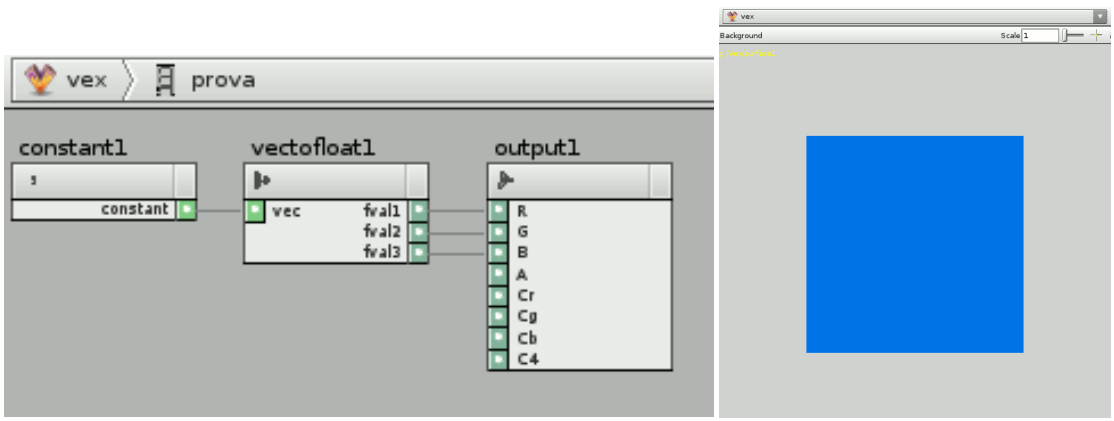


Figura 7: La xarxa VEX de l'esquerra produeix el resutat de la dreta

2.2.3 El node CONSTANT

Aquest node retorna un valor constant. És útil per crear valors que no hagin de canviar en les diferents instancies de la xarxa VEX. Aquests valors poden ser de tipus numèric, vector, matriu, string o color. En veiem un exemple en la Figura 8.

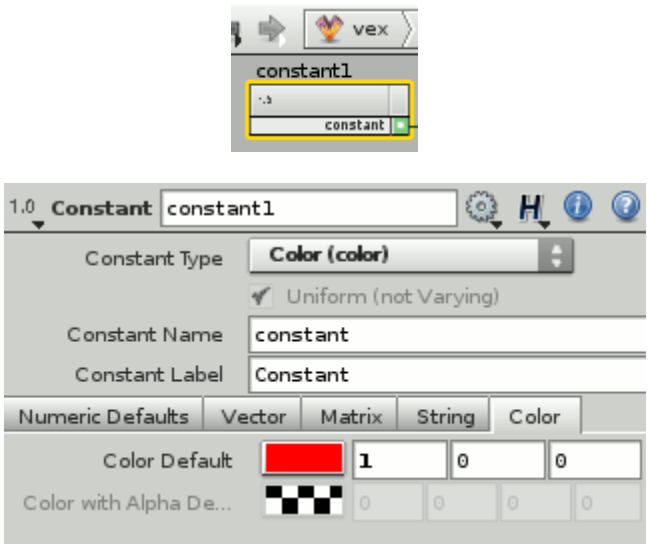


Figura 8: Exemple de node CONSTANT, on li definim una constant de tipus color.

2.2.4 El node COMPARE

Aquest node rep dos valors del mateix tipus i retorna un valor booleà amb el resultat de la comparació que li haguem especificat. També pot tenir només una entrada i li especifiquem manualment amb quin valor cal comparar-la. En veiem un exemple del seu ús en la Figura 9.

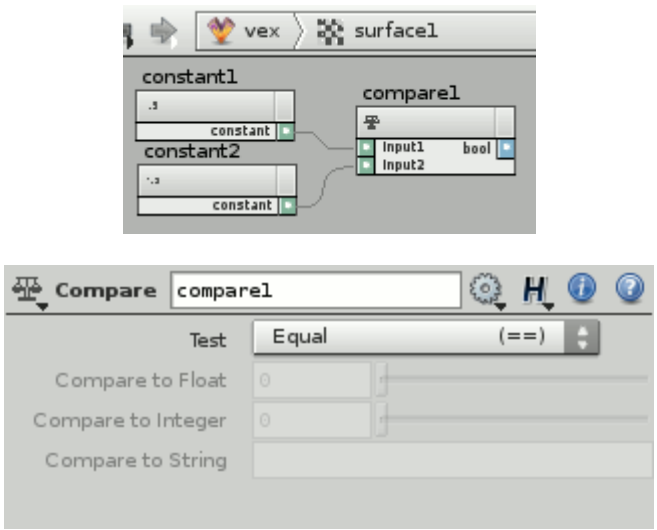


Figura 9: En aquest exemple, el node COMPARE evalua si els dos nodes CONSTANT tenen el mateix valor.

2.2.5 El node IF

Aquest node ha estat crucial per a diferents etapes dins del projecte, ja que ens permet definir dintre seu una subxarxa que només s'executarà si es compleix certa condició. El node IF

generalment vindrà precedit d'un node de tipus COMPARE (explicat a la secció 2.2.4), però també es pot fer servir qualsevol valor enter.

Qualsevol valor que es vulgui modificar dintre del node IF ha de ser connectat com a entrada. Les sortides del node IF contindran els valors modificats de les entrades (sempre que s'hagi complert la condició d'entrada). Els valors reals de les variables connectades mai es modifiquen, de manera que es poden connectar a altres nodes de la xarxa.

Dintre del node IF hi definirem una subxarxa VEX. Aquesta subxarxa tindrà com a mínim dos nodes: SUBINPUT i SUBOUTPUT. El node SUBINPUT tindrà disponibles com a sortides, les entrades que haguem endollat al node IF. Similarment, el node SUBOUTPUT definirà les sortides que tindrà el node IF, que seran les variables del SUBINPUT més les que haguem definit dintre la subxarxa i que vulguem fer servir després.

Veurem millor el funcionament amb un exemple senzill:

En la xarxa VEX principal, tenim dos nodes del tipus CONSTANT i un node COMPARE que comprovarà que ambdós tinguin el mateix valor (Figura 10).

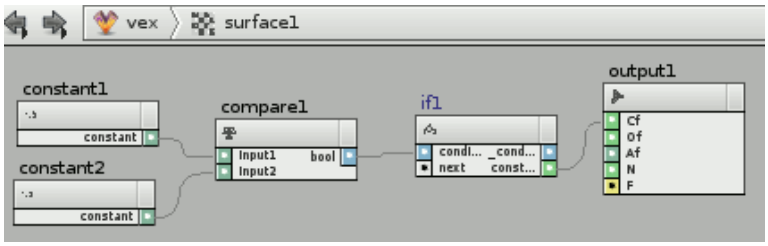


Figura 10: Exemple de xarxa VEX amb un node IF

En cas afirmatiu, s'executarà la subxarxa que conté el node IF, que podem veure en la Figura 11. Podem observar com subinput1 té com a sortida la condició que li hem endollat al node IF, i que suboutput1 té com a entrades aquesta mateixa condició i una constant que definim en la pròpia subxarxa, que són les sortides del node IF.

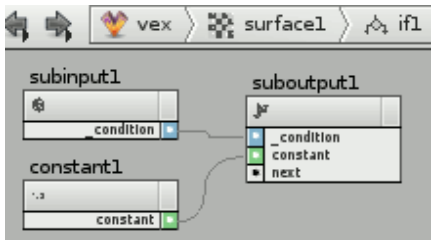


Figura 11: Exemple de subxarxa VEX dintre un node IF

No existeix la manera d'especificar què ha de fer el node en cas que no es compleixi la condició. Per aconseguir el mateix efecte que una sentència if-else, calen dos nodes IF precedits de dos nodes COMPARE amb condicions oposades (o bé un sol node COMPARE si un dels IF té el flag Condition posat a True i l'altre a False). Al primer IF cal connectar-li, a més, les

variables que ens calguin. A continuació, cal connectar totes les sortides del primer IF (menys la condició) com a entrades del segon IF. En veiem un exemple a la Figura 12.

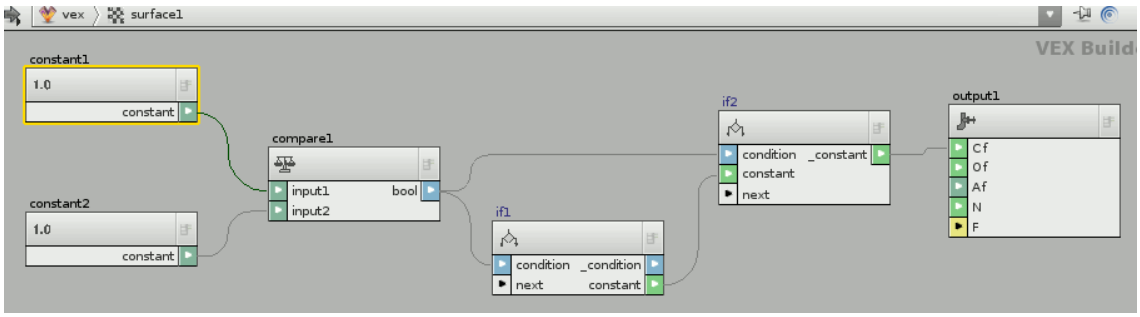


Figura 12: Exemple de sentència if-else

2.2.6 El node VORONOISE

El soroll de Voronoi funciona distribuint punts aleatòriament per l'espai d'acord amb una distribució de Poisson, generant patrons en forma de cel·les i retornant les posicions reals dels dos punts més propers. Per a una descripció més detallada, consultar la secció 3.2.1.

Figura 13: Exemple de node VORONOISE

2.2.7 El node STRIPES

Aquest node genera patrons de línies, i ens serà útil per a generar alguns patrons de les nostres ciutats. La sortida és el resultat d'una funció que evalua si cal pintar el punt actual, depenent de la freqüència i amplada que li passem com a paràmetres.

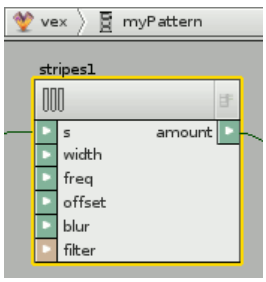


Figura 14: Node STRIPES

Si no se li especifica una entrada s, s'assumeix la variable global s. En la Figura 15 veiem el resultat de STRIPES amb amplada=0.3 i freqüència=5.

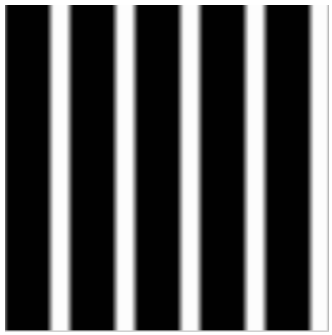


Figura 15: Resultat de STRIPES

2.3 Compositing (COP2)

Dintre la xarxa COP2 (o compositing) podem editar i aplicar filtres a imatges 2D. Aquestes imatges poden ser un llenç d'un color determinat, un fitxer extern o un *shader* creat en una xarxa VEX.

Per visualitzar els resultats d'aquest contexte, cal obrir una pestanya a la zona de treball del tipus Composite View. A continuació explicarem alguns nodes importants que farem servir en aquesta xarxa.

2.3.1 El node FILE

Amb aquest node importem un arxiu d'imatge que tinguem emmagatzemat al disc dur per poder manipular-lo després. El node prendrà automàticament el nom de l'arxiu d'imatge, com veiem a la Figura 16.

Figura 16: Node FILE que obtenim important l'arxiu aigua.png

2.3.2 El node VEX FILTER

Aquest node serveix per importar a la xarxa COP2 els resultats d'una xarxa VEX i així poder-ho manipular com a imatge bidimensional. El node prendrà automàticament el nom de la nostra xarxa VEX, com veiem a la Figura 17.

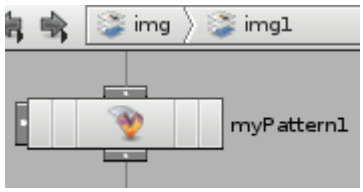


Figura 17: Node resultant d'importar la xarxa myPattern

2.3.3 El node MONO

Aquest node rep una imatge en color i la converteix en blanc i negre. Aquest pas serà necessari per més tard crear una geometria a partir d'aquesta imatge, ja que així ens desfem d'informació de color innecessària per al nostre projecte.

Figura 18: Exemple de node MONO

2.3.4 El node SCALE

Aquest node serà de gran utilitat quan haguem de treballar amb més d'una imatge i aquestes siguin de diferents mides i/o resolucions, ja que escala la imatge que rep com a primera entrada a la mida o resolució de la imatge que rep com a segona entrada. En podem veure un exemple a la Figura 19

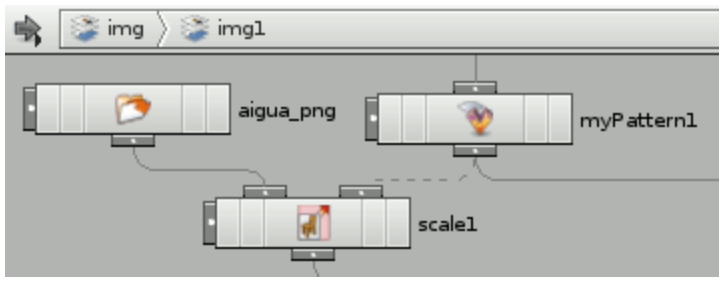


Figura 19: La sortida de scale1 serà aigua_png escalat a la mida de myPattern1

2.4 SOP (Geometria 3D)

En aquesta xarxa és on crearem i transformarem tota la geometria 3D. A continuació, per il·lustrar-ne el funcionament, es crea com a exemple una figura senzilla: un cub, que després es modificarà.

2.4.1 Exemple

➔ Primer pas: es crea un BOX

Per a crear un cub, senzillament es clica sobre el botó corresponent i el Houdini genera la figura. Simultàniament, s'obtenen dues representacions del mateix objecte, que veiem a la Figura 20. Per una banda, a la zona de treball s'hi genera la geometria en 3D i, per l'altra, es pot visualitzar l'escena en forma d'arbre de nodes, cadascun dels quals representa un objecte o acció.

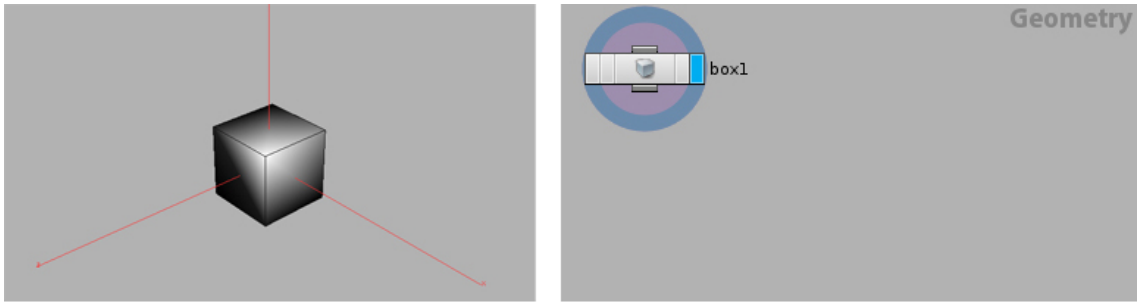


Figura 20: La geometria del cub i el seu respectiu node a l'arbre.

➔ Segon pas: s'aplica al BOX una transformació creant un node POLYEXTRUDE

A continuació, s'aplica a la figura una extrusió geomètrica creant, a partir de les 6 cares del cub, 6 cubs suplementaris. Aquest cop, usant l'arbre de nodes, es crea un POLYEXTRUDE i s'enllaça directament amb el BOX creat anteriorment. La sortida de dades del BOX es connecta amb l'entrada del POLYEXTRUDE, amb el resultat que veiem a la Figura 21.

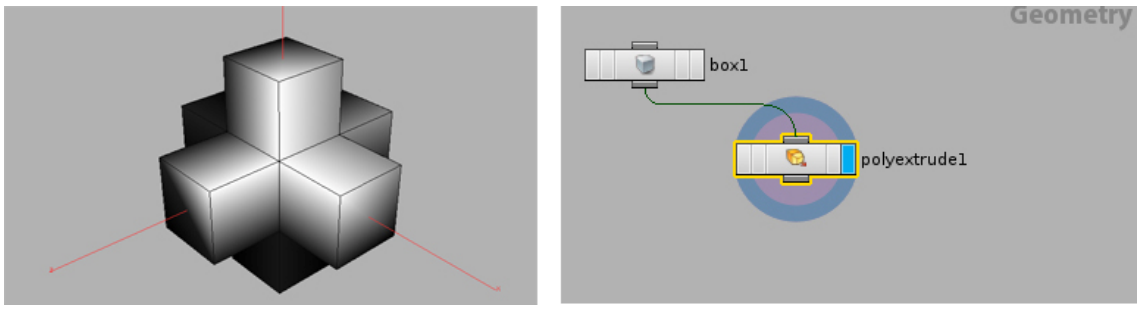


Figura 21: Per a crear una extrusió, cal afegir un node POLYEXTRUDE a l'arbre de nodes i enllaçar-lo amb el BOX.

➔ Tercer pas: s'aplica al sistema un rotació en tots els eixos creant un node TRANSFORM

S'aplica una rotació a la figura usant un node TRANSFORM. Com s'ha fet abans, es connecta la sortida del POLYEXTRUDE amb l'entrada del TRANSFORM, obtenint el resultat que veiem a la Figura 22.

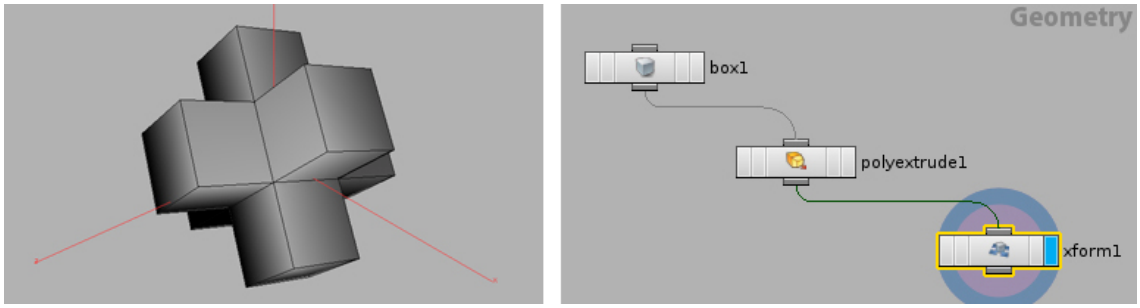


Figura 22: De la mateixa manera que al punt anterior, per a rotar l'objecte, s'aplica un node TRANSFORM i s'enllaça amb el POLYEXTRUDE, formant una cadena.

En general, cadascun dels nodes representa una funció que té una entrada de dades, manipula aquestes dades i retorna un resultat. Amb aquest sistema, s'obté un conjunt d'objectes independents enllaçats entre ells a través d'un flux de dades.

➔ Un exemple: es canvia el BOX per un TORUS

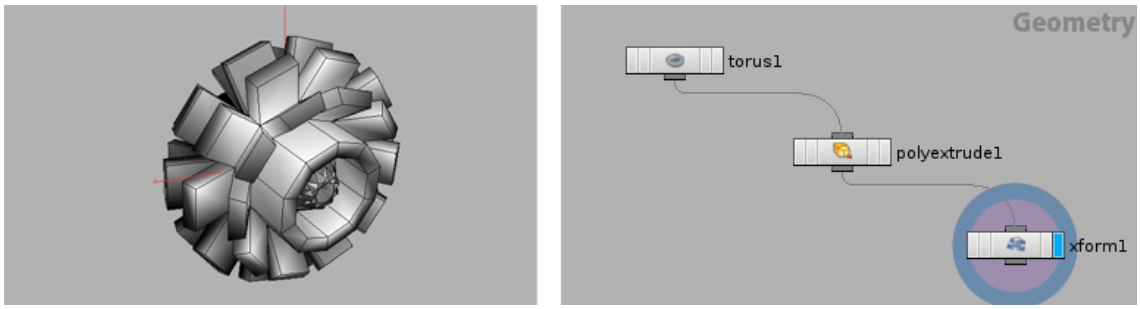


Figura 23: El canvi de l'objecte de base no altera el funcionament dels altres nodes. Senzillament s'obté un resultat diferent.

En qualsevol moment del procés, es pot fer un canvi en un node inicial i aquest s'anirà propagant a través dels seus successors. De la mateixa manera, es pot aplicar una transformació en qualsevol punt i s'obté un nou flux procedural usant els mateixos nodes de base.

2.4.2 Enumeració de primitives - divisions

Molts objectes de Houdini poden ser dividits en primitives enumerades, quelcom significatiu dins aquest projecte. Cal poder enumerar i diferenciar els diferents objectes que anem creant, cosa que ja fa automàticament el Houdini, com es veu a la Figura 24.

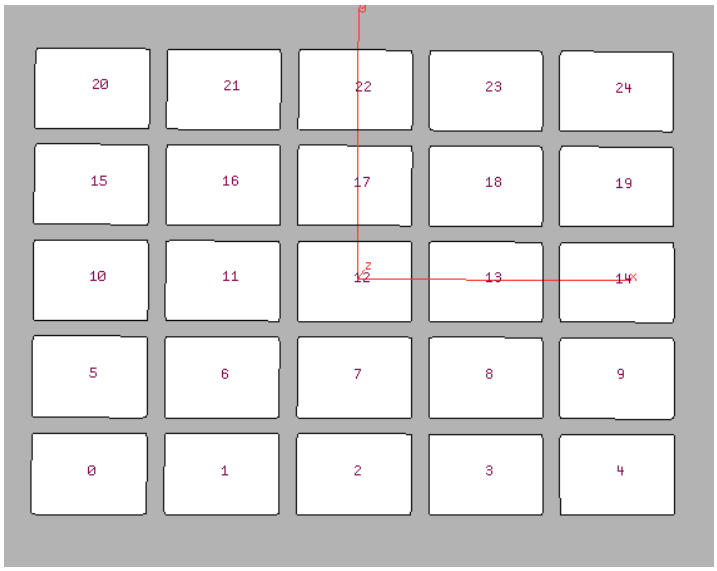


Figura 24: Totes les divisions són enumerades i referenciables.

2.4.3 Node DELETE

Aquest node juga un paper molt important dins la implementació del programa. La seva funció bàsica és eliminar geometria. La Figura 25 mostra un node de tipus GRID amb N divisions. Aquestes divisions estan enumerades i, per tant, poden ser referenciades. Tenint en compte això, a la Figura 26 s'ha aplicat un node DELETE que, en aquest cas, elimina una de cada dues divisions de 0 a N. La Figura 27 mostra el tauler de propietats del node DELETE. Es pot veure les variables modificades per tal que s'eliminin una de cada dues primitives de 0 fins a $N. La variable $N és la referència que fa servir Houdini per a representar el nombre màxim de primitives d'un objecte.

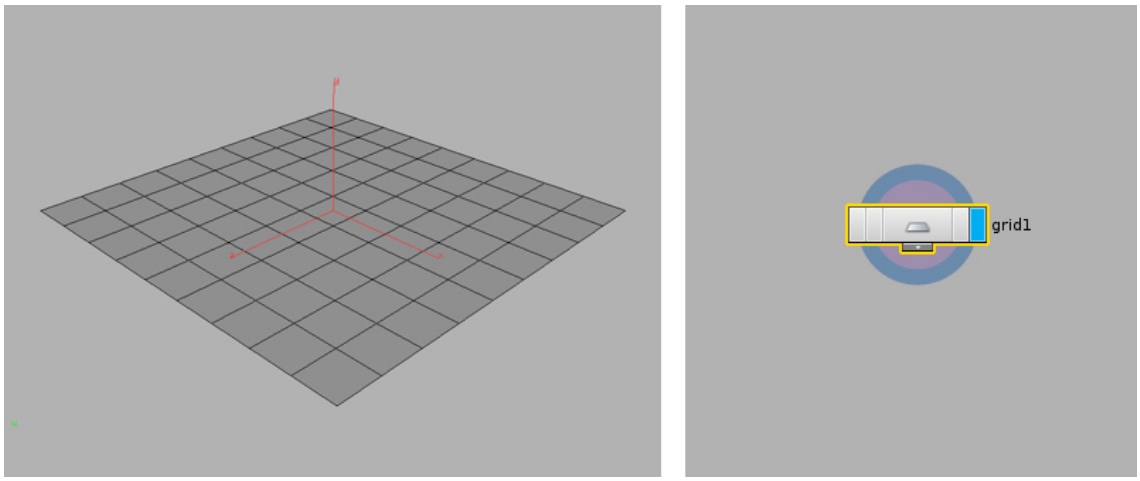


Figura 25: S'ha creat un node de tipus GRID.

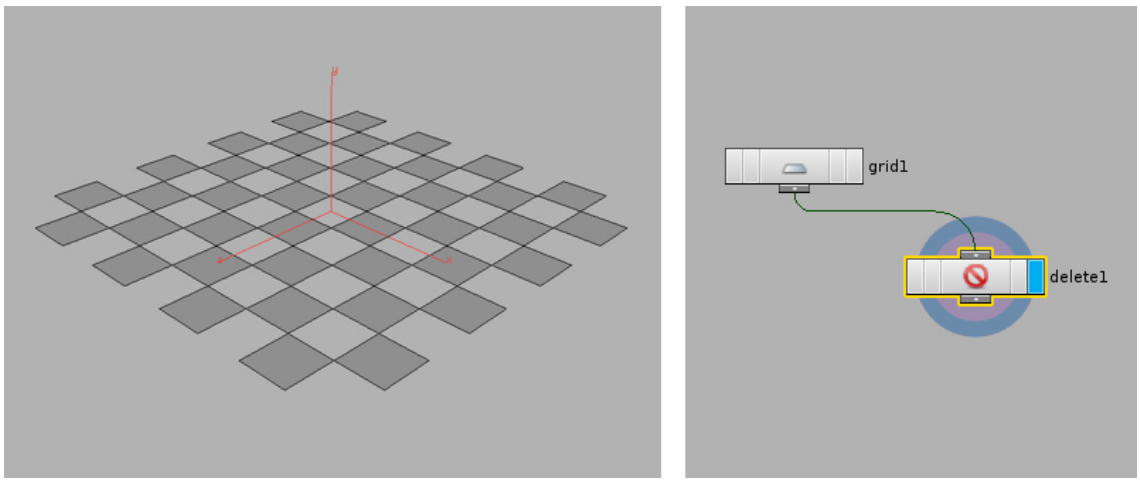


Figura 26: Un cop aplicat el DELETE, s'han eliminat una de cada dues primitives, de 0 a N.

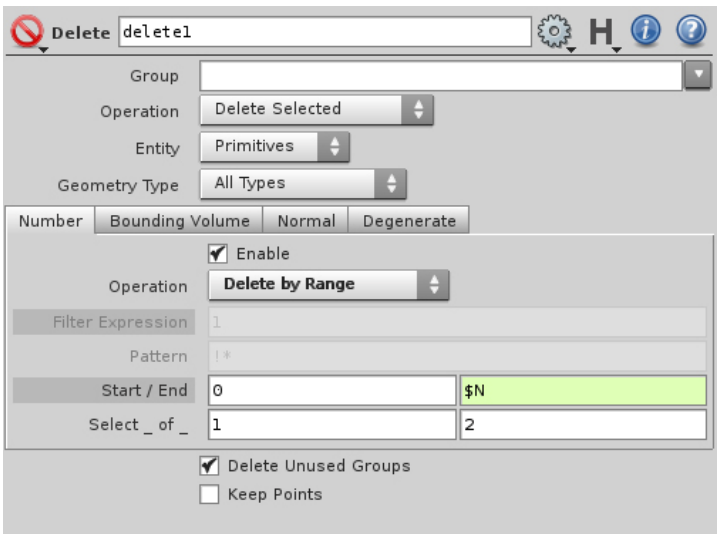


Figura 27: Propeitats del node DELETE. En aquest cas s'eliminen una de cada dues primitives seleccionades dintre del rang 0..N.

2.4.4 El node COPY

El node COPY permet, entre altres coses, aplicar una o vàries accions sobre diferents primitives. Normalment, el flux de dades avança endavant en l'arbre de nodes. Amb el node COPY podem tornar enrere i repetir una sèrie d'accions tantes vegades com primitives tinguem, creant una mena de bucle.

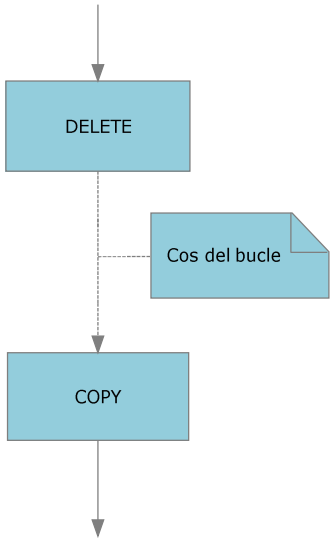


Figura 28: Esquema del bucle creat pels nodes DELETE i COPY

Cal definir-li una variable que farem servir després en una operació anomenada "stamp" (segell), que prendrà el valor del número de primitiva actual (valor que conté la variable local del Houdini $CY), com mostra la Figura 29.

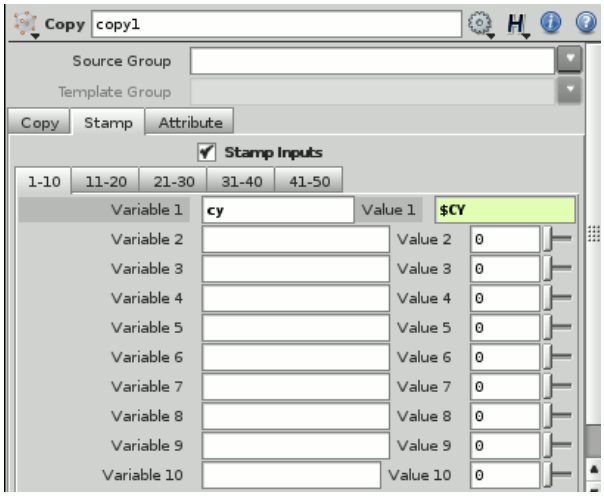


Figura 29: Propeitats del node COPY. Definint una variable anomenada cy amb el valor de la primitiva actual.

El bucle començarà amb un node DELETE. La Figura 30 mostra el seu panell de propietats, on li indicarem al paràmetre group que cal eliminar totes les primitives menys una cada vegada. Això ho farem amb una funció "stamp", que té com a paràmetres la localització del node COPY, el nom de la variable que hem definit al node COPY i un valor per defecte.

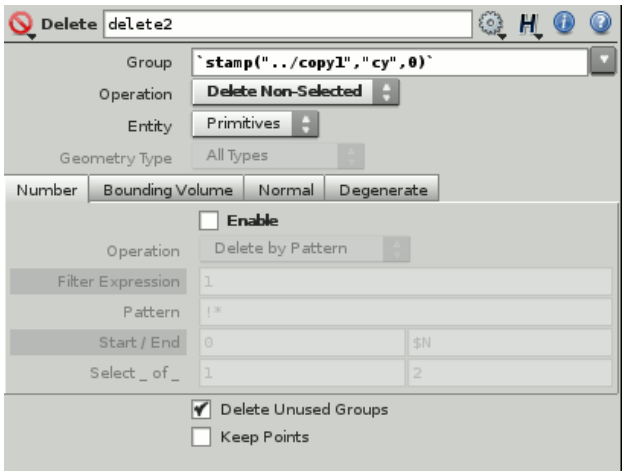


Figura 30: Propeitats del node COPY. Definint una variable de stamp anomenada cy amb el valor de la primitiva actual.

D'aquesta manera, tot el que hi hagi entre el node DELETE i el node COPY s'aplicarà per totes les primitives.

En la Figura 31 veiem un exemple on a s'ha aplicat una rotació d'un angle aleatori a cada primitava mitjançant un node TRANSFORM, que ens permet fer diferents transformacions.

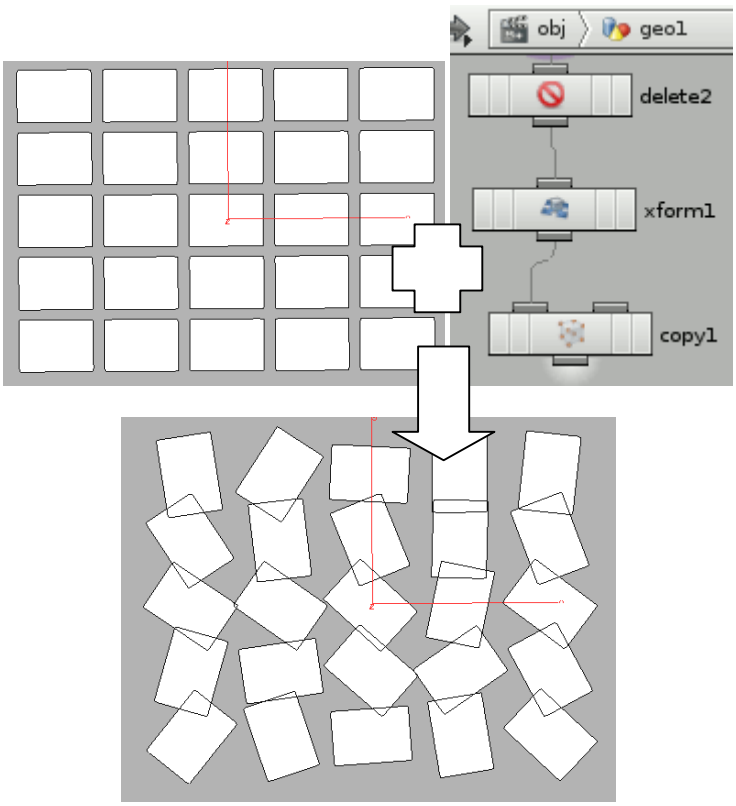


Figura 31: Cada primitiva ha rotat un angle diferent

2.4.5 El node CREEP

Aquest node serveix per "deformar" un tros de geometria per fer-lo encaixar en una superfície. En el nostre cas, el farem servir per encaixar els edificis en el seu espai.

En l'exemple de la Figura 32, tenim un node CREEP on la primera entrada és un BOX i la segona porta el flux de dades d'una superfície plana:

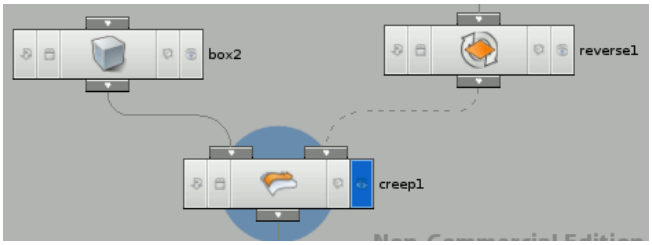


Figura 32: El node creep1 deformarà box2 en la forma de reverse1

El resultat, com veiem a la Figura 33, és que el BOX s'ha adaptat a la forma de la superfície:

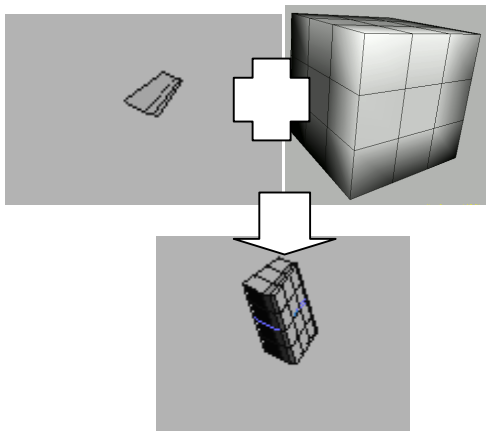


Figura 33: El box s'ha adaptat a la superficie

2.5 Houdini Digital Asset

Els Houdini Digital Assets (HDA) són nodes personalitzats construits a partir de xarxes de nodes ja existents. S'encapsula la xarxa en un HDA i després es "promocionen" paràmetres de dintre l'asset com a paràmetres d'aquest nou nivell més alt. És equivalent a definir una nova classe en programació orientada a objectes però, en aquest cas, a nivell geomètric.

D'aquesta manera podem encapsular tota la xarxa SOP en un sol node, de manera que pot funcionar com una interfície d'usuari que tingui només aquells paràmetres que l'usuari hagi de manipular.

Per crear-ne un, cal seleccionar els nodes a encapsular i premer SHIFT+C per crear una subnet (subxarxa). A continuació, cal fer click amb el botó dret sobre el subnet i triar la opció *Create Digital Asset,* moment en el que s'obrirà una finestra per a configurar-lo. Des de la pestanya Parameters, que veiem a la Figura 34, es pot configurar quins paràmetres volem per al nostre HDA arrossegant-los de la llista de tots els paràmetres de tots els nodes encapsulats.

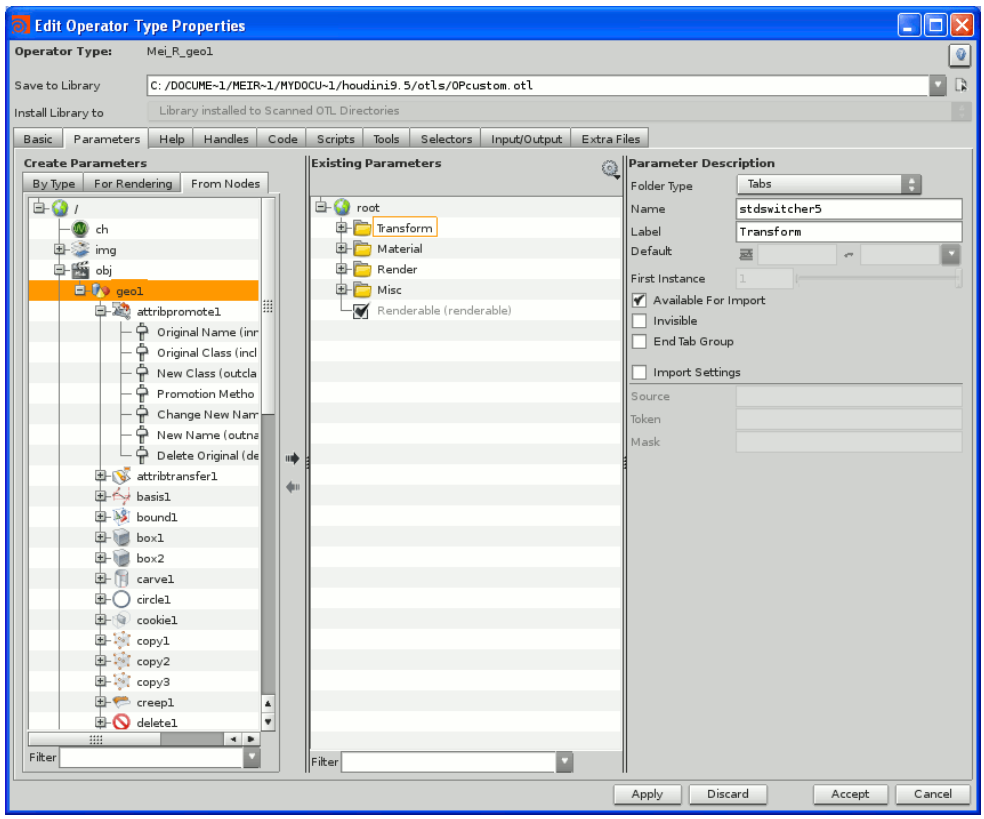


Figura 34: Des d'aquesta pantalla podem triar quins paràmetres tindrà el nostre HDA

2.6 Llenguatges d'scripting

Houdini incorpora per defecte dos llenguatges d'script dins el sistema. El primer d'ells, i més antic, és l'anomenat HSript. Aquest llenguatge és exclusiu de Houdini i existent des de les primeres versions. El segon és el Python, introduït a partir de la versió 9.0 amb l'objectiu d'estandarditzar el funcionament i dotar el programa d'un codi conegut.

S'ha decidit utilitzar el Python perquè està present a la versió més recent del Houdini, perquè és més potent que l'HScript i perquè posa a la disposició de l'usuari no només les funcions de Houdini, sinó totes les llibreries existents de Python.

Python és un llenguatge de programació creat per Guido van Rossum en l'any 1990. Es compara habitualment amb TCL, Perl, Scheme, Java i Ruby. En l'actualitat Python es desenvolupa com un projecte de codi obert, administrat per la Python Programari Foundation. L'última versió estable del llenguatge és actualment la 3.0.1 (13 de febrer de 2009) [6].

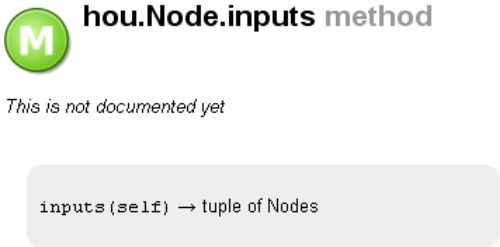
Python és considerat com la "oposició lleial" a Perl, llenguatge amb el qual manté una rivalitat amistosa. Els usuaris de Python consideren a aquest molt més net i elegant per a programar. Python permet dividir el programa en mòduls reutilitzables des d'uns altres programes Python. També hi ha mòduls inclosos que proporcionen E/S de fitxers, crides al sistema, sockets i fins a interfícies a GUI (interfície gràfica amb l'usuari) GTK, Qt entre altres...

Python és un llenguatge interpretat, el que estalvia un temps considerable en el desenvolupament del programa, perquè no és necessari compilar ni enllaçar. L'intèrpret es pot utilitzar de manera interactiva, el que facilita experimentar amb característiques del llenguatge, escriure programes d'un sol ús o provar funcions durant el desenvolupament del programa.

El principal objectiu que persegueix aquest llenguatge és la facilitat, tant de lectura, com de disseny.

2.7 Python dins de Houdini

Actualment encara no està acabada tota la implementació de Python dins de Houdini. Això fa que la majoria de vegades sigui impossible trobar documentació sobre les funcions i procediments que disposa el programa.



2.8 Usant Python

Primer de tot cal aclarir les dues maneres de fer servir Houdini. La primera, i més habitual, és utilitzant la interfície d'usuari del programa. La segona és fent servir la consola de Python. Usant la consola es poden fer totes les accions disponibles a la interfície gràfica.

Com que Python és un llenguatge interpretat, es pot executar funcions en temps real, tal i com es faria prement un botó del programa.

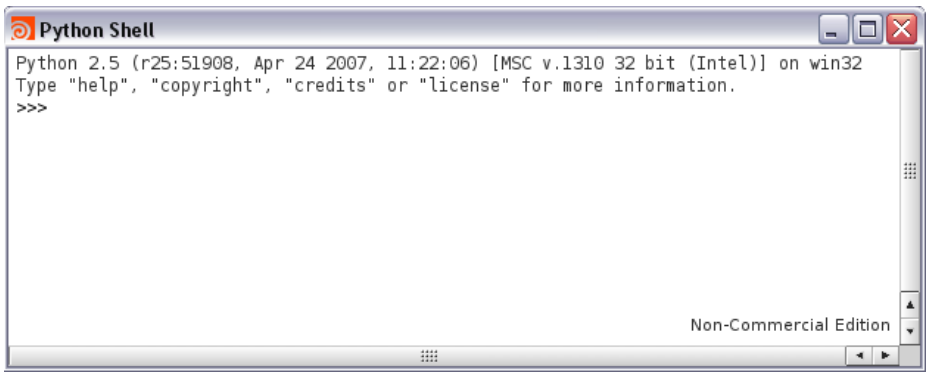


Figura 35: Captura de la consola de Python. Per accedir-hi: Windows > Python Shell.

El sistema Python que té Houdini és exactament el mateix que es pot descarregar des de la web oficial de Python. L'única diferència és la incorporació d'un mòdul addicional anomenat ***hou*** que conté totes les funcionalitats, objectes i propietats de Houdini. Aquesta API rep el nom de HOM (*Houdini Object Model*).

Un exemple creant un BOX amb la consola de Python. Primer de tot es s'importa el mòdul *hou*. Seguidament es crea un objecte de tipus *'geo'* que emmagatzemarà tots els objectes. Finalment es crea el box utilitzant al sentència *"createNode()"*.

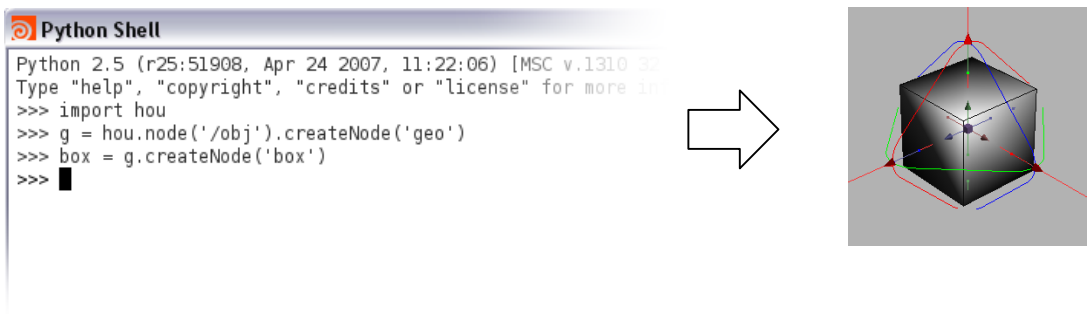


Figura 36: Creació d'un cub utilitzant el Python.

També es poden aplicar transformacions a través de la consola. En aquest cas es crea un node POLYEXTRUDE i s'enllaça amb el BOX utilitzant la sentència "*setFirstInput()*". Llavors es canvia el paràmetre *tz* del node POLYEXTRUDE.

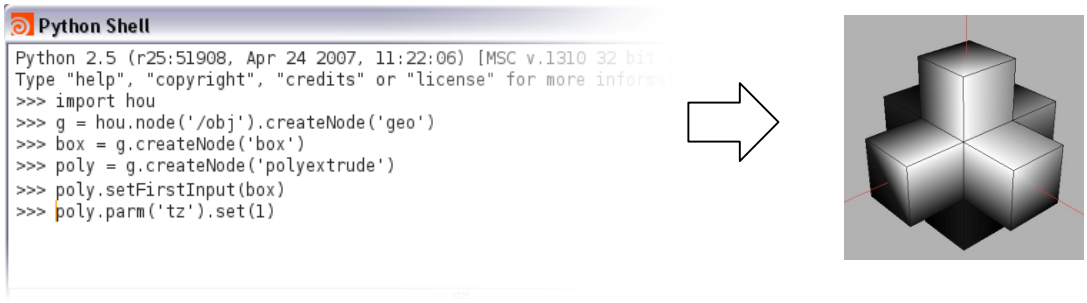


Figura 37: També es poden aplicar transformacions fent servir la consola.

També és possible carregar scripts des de fitxers externs. Aquesta és la manera que s'ha triat per a executar el projecte.

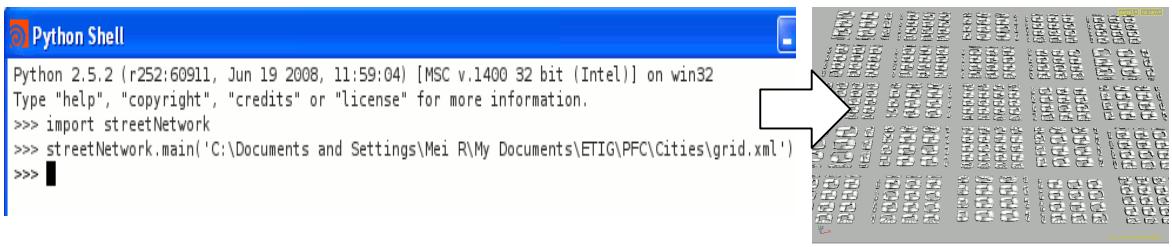


Figura 38: Des de la consola és des de on es carreguen els scripts externs.

Un cop s'ha importat un script aquest queda desat en memòria. Si es fa algun canvi al fitxer de l'script, només cal executar la sentència *reload(mòdul)* i s'actualitzarà la càrrega.

Cada vegada que es crea un objecte dins de Houdini aquest rep un nom que apunta a un espai de memòria on hi ha desades les dades de l'objecte. Quan aquesta creació es fa des de la interfície gràfica, aquest nom és invisible a l'usuari, que només pot veure l'etiqueta que rep l'objecte. En canvi, quan l'objecte és creat des de Python, la manera d'accedir a aquest és mitjançant el nom. En aquest cas, l'etiqueta es comporta com una propietat més, accessible a través dels mètodes apropiats. Aquesta etiqueta pot ser modificada, però amb la restricció que ha de ser única.

2.9 Llenguatge per al disseny de xarxes de carrers

Arribats a aquest punt, l'únic problema que quedava per resoldre era trobar un llenguatge suficientment versàtil, senzill i conegut per a usar-lo com a llenguatge de disseny per a l'usuari final.

Tot i que es podria haver desenvolupat un llenguatge propi, es va optar per fer servir XML. Per a prendre aquesta decisió es van tenir en compte diferents factors, com ara que el Python incorpora per defecte un parser d'XML, el format standard del llenguatge o la possibilitat d'incorporar un sistema de validació (DTD, XML Schema) de manera relativament senzilla.

XML, de l'anglès eXtensible Markup Language, és un metallenguatge extensible, d'etiquetes, desenvolupat pel World Wide Web Consortium [8].

És una simplificació i adaptació de l'SGML, i permet definir la gramàtica de llenguatges específics (de la mateixa manera que HTML és, alhora, un llenguatge definit per SGML). Per tant, XML no és realment un llenguatge en particular, sinó una manera de definir llenguatges per a diferents necessitats. Alguns dels llenguatges que empren XML per a la seva definició són XHTML, SVG, MathML.

XML no ha nascut només per a la seva aplicació a Internet, sinó que es proposa com a un estàndard per a l'intercanvi d'informació estructurada entre diferents plataformes. Es pot utilitzar per a bases de dades, editors de text, fulls de càlcul i per moltes altres aplicacions diverses.

XML és una tecnologia relativament senzilla que té al seu voltant altres que la complementen i la fan notablement més extensa, a més de proporcionar-li unes possibilitats molt més grans. A l'actualitat té un paper molt important, ja que permet la compatibilitat entre sistemes, permetent de compartir informació d'una manera segura, fiable i fàcil.

En aquest cas, s'utilitzarà l'XML com una eina de desenvolupament que donarà a l'usuari la possibilitat de crear models de ciutats en 3D d'una manera ràpida i fàcil. El fet que aquest llenguatge sigui estàndard i conegut és un punt a favor a l'hora d'aprendre a fer servir el programa. La sintaxi senzilla i la quantitat d'informació existent sobre XML fa que requereixi un aprenentatge lleuger, fet que facilita la utilització del sistema.

El parser d'XML que incorpora el Python ha facilitat molt la feina a l'hora d'interpretar els fitxers XML's i transformar-los en una estructura de dades accessible des de Python.

Mitjançant aquest parser, s'obté una estructura de dades en forma d'arbre que segueix l'estàndard marcat per el DOM (*Document Object Model*)[7]. Així doncs, un cop recuperada l'estructura de l'XML i usant els mètodes que proporcionen les llibreries de Python, es pot accedir a tota la informació necessària per a construir la geometria.

Capítol 3: Treball previ

Dins aquest capítol s'introduiran tots aquells conceptes i referències que han servit d'inspiració per a realitzar el projecte.

3.1 CGA shape

El projecte s'ha basat sobretot en l'article *Procedural Modeling of Cities* (Müller & Parish, Procedural Modeling of Cities, 2001). L'objectiu de Müller i els seus companys és crear un sistema per a construir ciutats de manera automàtica i amb uns resultats realistes. Usant una sintaxi anomenada *CGA shape* i seguint les regles introduïdes amb aquesta sintaxi, s'aconsegueix crear la geometria d'una ciutat de manera ràpida i fàcil.

El CGA shape és una sintaxi basada en regles que serveix per a dissenyar l'estructura de la xarxa de carrers. Aquesta sintaxi està basada en els L-Systems (Lindenmayer, 1991), que s'utilitzen per a descriure el creixement de les plantes.

Les regles del CGA shape segueixen sempre un mateix patró:

Id: predecessor: condició → *successor: probabilitat*

on

- **Id** és un enter identificatiu de cadascuna de les regles.
- **predecessor** és aquell element que ha de ser substituït o sobre el qual s'aplica la regla.
- **condició** és el requeriment que s'ha de complir per a executar la regla.
- **successor** és el conjunt d'instruccions a executar.
- **probabilitat** és el coeficient de probabilitat que té la regla per ser aplicada.

3.2 Patrons de xarxes de carrers

Les xarxes de carrers de les ciutats formen patrons de gran complexitat. Aquest patrons estan condicionats per motius geogràfics (rius, llacs, elevacions del terreny), històrics i culturals. Les carreteres de les zones més habitades també haurien de ser més denses per alleujar la càrrega del trànsit, i la gent de tot arreu hauria de tenir un accés fàcil a les carreteres.

3.2.1 Patró voronoi

S'ha observat que moltes ciutats responen a un patró semblant als diagrames de Voronoi, com São Paolo (Figura 39) o Ciutat del Cap (Sun, Baciu, Yu, & Green, 2002).

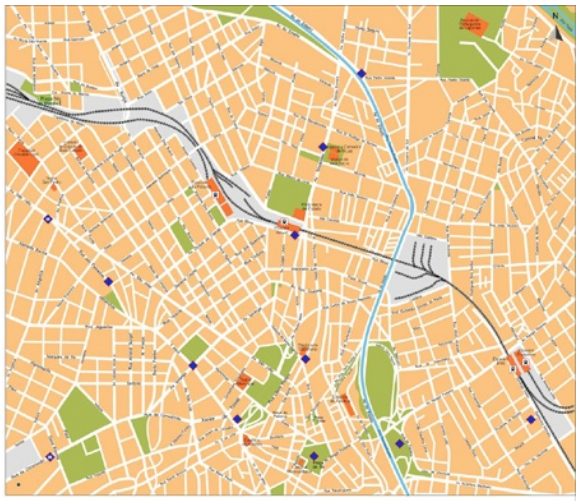
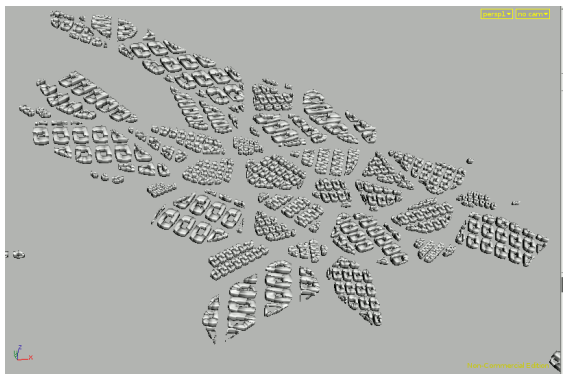


Figura 39: Mapa de São Paolo, que té una distribució semblant a un diagrama de Voronoi

És per això i per la versemblança dels resultats obtinguts que farem servir diagrames de Voronoi per a generar ciutats sintètiques.



En matemàtiques, un diagrama de Voronoi és un tipus especial de descomposició d'un espai mètric determinada per distàncies a una col·lecció discreta d'objectes a l'espai, per exemple, en el nostre cas, per una col·lecció discreta de punts.

En el cas més simple, ens donen una col·lecció de punts *S* en el pla, que són els punts Voronoi. Cada punt *s* ϵ S té associada una cel·la Voronoi *V(s)*, que consisteix en tots els punts del pla que són més a prop de *s* que de qualsevol altre punt de S. Els segments del diagrama de Voronoi són tots els punts del pla que són equidistants a dos punts de Voronoi. En la Figura 40 podem veure un exemple de diagrama de Voronoi, amb les diferents cel·les resultants pintades de diferents colors.

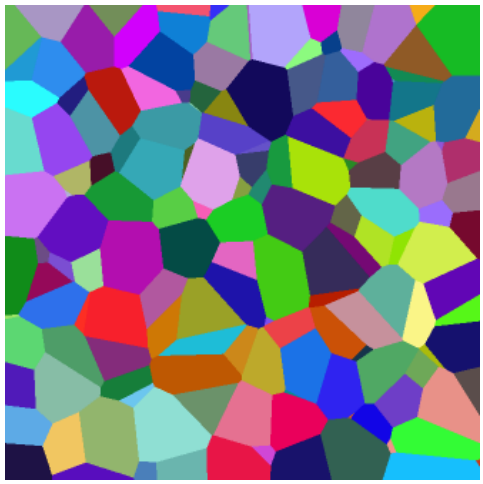


Figura 40: Exemple de diagrama de Voronoi

Per exemple, per la col·lecció de punts (*x*,*y*) amb *x* en una col·lecció discreta *X* i *y* en una col·lecció discreta *Y*, aconseguim "rajoles" rectangulars amb els punts no necessàriament als seus centres.

El node VORONOISE del Houdini (explicat a la secció 2.2.6) fa servir una variació dels diagrames de Voronoi anomenat algorisme de Lloyd, també anomenat iteració o relaxació de Voronoi.

En aquest algorisme, una vegada obtingut el diagrama de Voronoi, es calcula el punt mitjà (centroide) de cada cel·la *V(s)* i mou *s* a aquest lloc. Es van repetint aquests passos fins que obtenim un resultat suficientment bo, és a dir, quan la distribució ja sigui suficientment regular per als nostres propòsits.

```
MENTRE la distribució no sigui suficientment regular FER
      Computar el diagrama de Voronoi per a S
      Computar els centroides de cada cel·la
      Moure cada punt s al seu centroide
FMENTRE
```

Figura 41: Pseudocodi per al algorisme de Lloyd

A cada iteració, els punts queden en una distribució lleugerament més regular: els punts propers s'allunyen i els espaiats s'apropen. Això produeix diagrames de Voronoi centroidals, que produeixen una distribució més uniforme de punts, com veiem a la Figura 42.

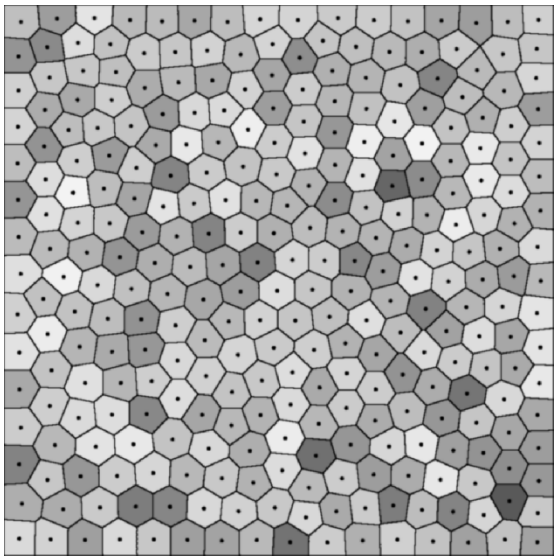


Figura 42: Diagrama de voronoi centroidal

El problema més gran que ens presenta l'ús dels diagrames de Voronoi pel propòsit de simular els carrers d'una ciutat és que la freqüència dels punts (i cel·les) resultants és uniforme en tot el pla, mentre que a la realitat els carrers no es distribueixen uniformement. Llavors, variem les posicions dels centroides variant la distribució de pesos a l'espai.

Per fer una aproximació una mica millor, farem servir una distribució de densitat radial, és a dir, volem que la densitat de població (i de carrers) sigui més gran al centre de la ciutat i que vagi disminuint a mesura que ens acostem a la perifèria, com es mostra a la Figura 43. Per fer-ho, calculem la distància de cada punt amb el centre, en funció de la qual determinem la freqüència.

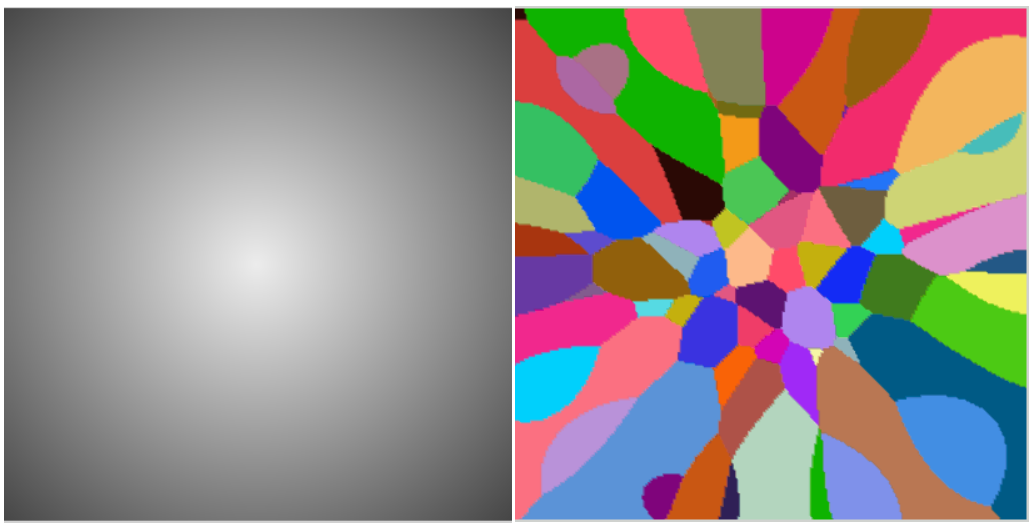


Figura 43: El mapa de densitat centrada de l'esquerra produeix el diagrama de Voronoi de la dreta

Alternativament, també podem utilitzar un mapa de densitats de població, on els tons més clars indiquen les zones més denses i els més foscos les menys poblades. Veiem un exemple d'això a la Figura 44.

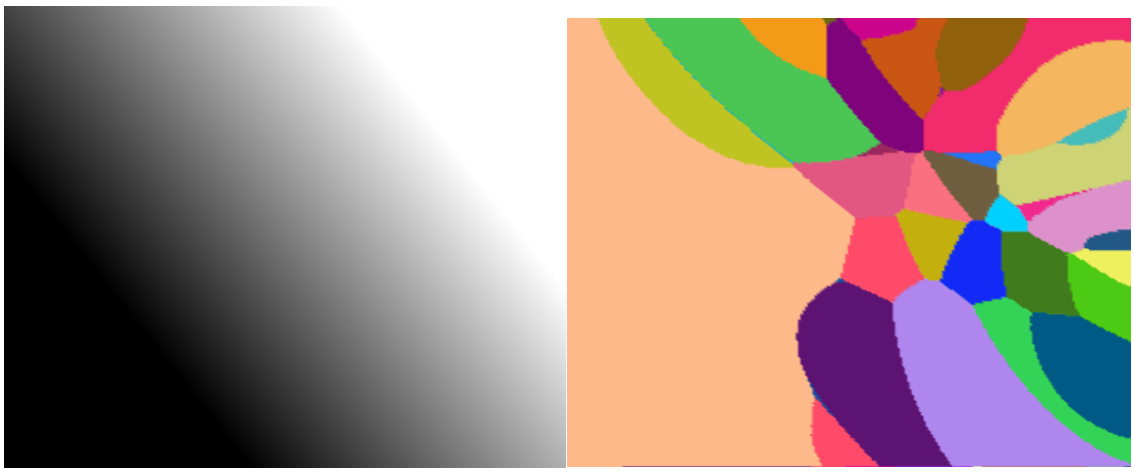


Figura 44: El mapa de densitats de l'esquerra produeix el diagrama de Voronoi de la dreta

Podem veure el diagrama d'activitat corresponent a aquest patró a la secció 4.2.2.

3.2.2 Patró grid

El patró grid (graella) és el més habitual en àrees urbanes, on carrers i carreteres formen blocs rectangulars. Sovint es troba aquest patró en districtes que són fruit de la concentració extraordinària de la població, la indústria i el trànsit a les ciutats, com passa a Nova York, Toronto o San Francisco. Un exemple proper seria l'Eixample de Barcelona (Figura 45), que va sorgir al segle XIX després de l'enderrocament de les muralles (1854-1856) i l'expansió de la ciutat.

Figura 45: A l'esquerra, l'Eixample de Barcelona. A la dreta, un exemple de ciutat sintètica amb patró grid

Podem veure el diagrama d'activitat corresponent a aquest patró a la secció 4.2.1.

3.2.3 Patró radial

El patró radial es dóna sobretot en ciutats que tenen un centre històric i que han anat creixent circularment al seu voltant. Alguns exemples són París i Karlsruhe (Figura 46).

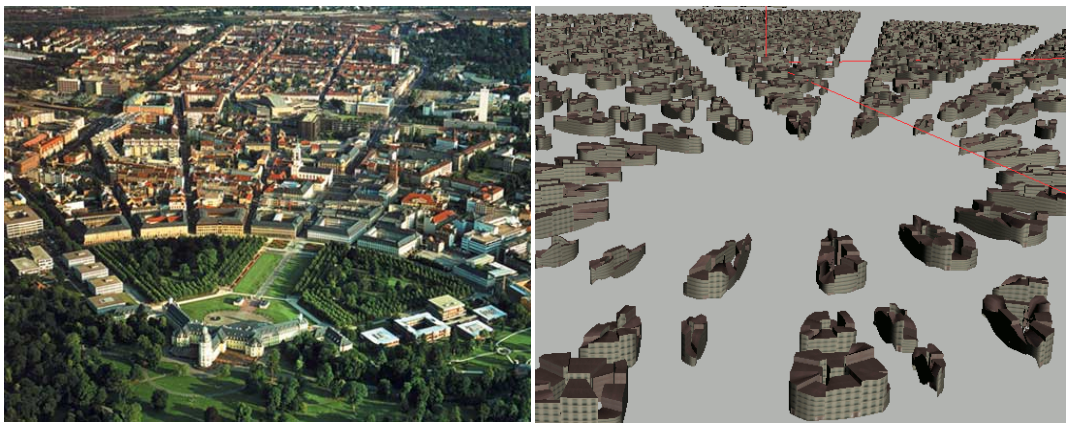


Figura 46: A l'esquerra, la ciutat alemanya de Karlsruhe. A la dreta, un exemple de ciutat sintètica de patró radial

Podem veure el diagrama d'activitat corresponent a aquest patró a la secció 4.2.1.

3.2.4 Districtes

Una ciutat real rarament consistirà d'un sol patró. Generalment, al llarg dels anys, les ciutats creixien i evolucionen per acabar consistint en diferents districtes o barris amb les seves estrucutres i característiques individuals. Per exemple, moltes ciutats tenen un barri vell, que tindrà una estructura molt diferent a les zones més noves.

Podem veure el diagrama d'activitat per a generar diferents patrons segons els districtes a la secció 4.2.3.

Capítol 4: Anàlisi

En aquest capítol es definiran els requeriments del sistema. Per a aquest objectiu, es farà servir el Unified Modeling Language (UML), que és un llenguatge de modelatge estàndard per al camp de l'enginyeria del programari.

4.1 Diagrames de casos d'ús

Un cas d'us és una tècnica per a la captura de requisits potencials d'un nou sistema. Cada cas d'ús proporciona un o més escenaris que indiqun com hauria d'interactuar el sistema amb l'usuari o amb un altres sistema per a aconseguir un objectiu específic. Normalment, als casos d'ús s'evita utilitzar llenguatge tècnic, preferint un llenguatge més proper a l'usuari final.

4.1.1 Diagrama de cas d'ús general

Aquest diagrama representa l'escenari més general amb que es troba l'usuari, que és la decisió entre generar una ciutat real o bé una de sintètica.

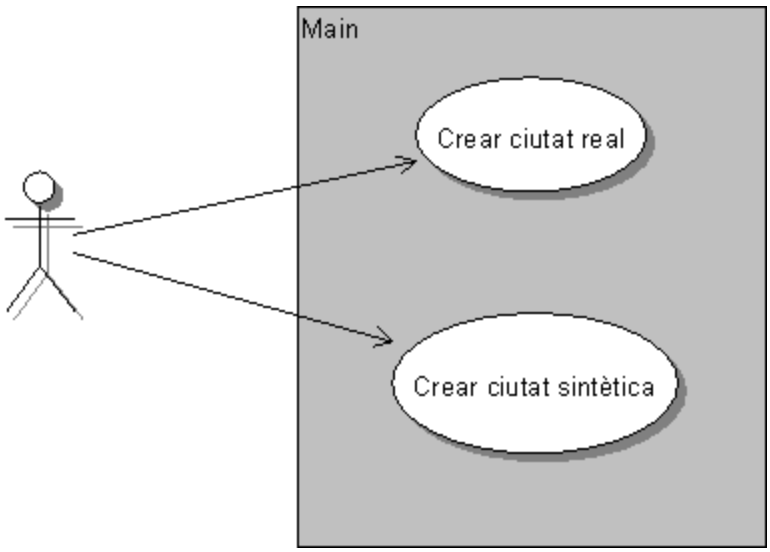


Figura 47: Diagrama de cas d'ús general

4.1.1.1 Fitxes cas d'ús general

FITXA	Crear ciutat real
Funcionalitat	A partir d'aquest cas d'ús es crearà una ciutat a partir d'un mapa d'una ciutat real.
Actor(s)	Usuari.
Pre-condició	Cap.
Flux principal	1. Carregar mapa. 2. Crear parcel·les. 3. Aixecar edificis.
Subfluxos (exensions)	Cap.
Flux alternatiu	Cap.
Post-condició	S'ha creat una ciutat a partir del mapa donat com a entrada.

FITXA	Crear ciutat sintètica
Funcionalitat	A partir d'aquest cas d'ús es crearà una ciutat sintètica a partir dels paràmetres entrats per l'usuari.
Actor(s)	Usuari.
Pre-condició	Cap.
Flux principal	1. Determinar patró. 2. Determinar ús de la terra. 3. Crear carrers. 4. Crear parcel·les. 5. Aixecar edificis.
Subfluxos (exensions)	• En cas d'haver-hi mapa de districtes, dibuixar patrons en conseqüència. • En cas que el patró sigui voronoise, corregir densitat.
Flux alternatiu	Cap.
Post-condició	S'ha creat una ciutat a partir del mapa donat com a entrada

4.1.2 Cas d'ús per a ciutat real

Aquest diagrama representa les diferents paràmetres que l'usuari pot triar a l'hora de generar una ciutat real, que són el mapa de la ciutat (i, opcionalment una textura per al terra), les opcions per a la divisió en parcel·les (amplada de la parcel·la) i el path on tenim guardades les textures per als edificis.

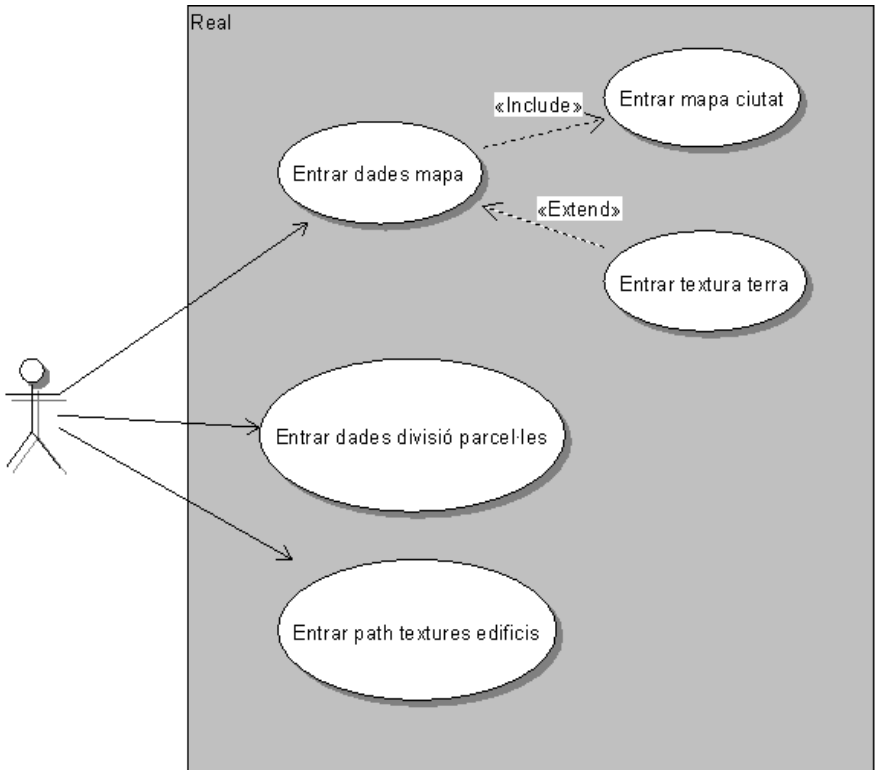


Figura 48: Diagrama de cas d'ús per a ciutat real

4.1.2.1 Fitxes cas d'ús ciutat real

FITXA	Entrar dades mapa
Funcionalitat	L'usuari entra les dades necessàries per a carregar el mapa.
Actor(s)	Usuari.
Pre-condició	Cap.
Flux principal	1. L'usuari entra el path del mapa a utilitzar per carregar la ciutat.
Subfluxos (exensions)	• L'usuari pot opcionalment entrar el path per a posar una textura al terra.
Flux alternatiu	Cap.
Post-condició	Tenim les dades per a crear la geometria bàsica.

FITXA	Entrar dades divisió parcel·les
Funcionalitat	L'usuari entra les dades necessàries per a dividir els blocs en parcel·les.
Actor(s)	Usuari.
Pre-condició	Cap.
Flux principal	1. L'usuari entra l'ampada de les parcel·les
Subfluxos (exensions)	Cap.
Flux alternatiu	Cap.
Post-condició	Tenim les dades per a crear les parcel·les.

FITXA	Entrar path textures edificis
Funcionalitat	L'usuari entra les dades necessàries per a posar textures als edificis.
Actor(s)	Usuari.
Pre-condició	Cap.
Flux principal	1. L'usuari entra el path on es troben les textures que tindran els edificis.
Subfluxos (exensions)	Cap.
Flux alternatiu	Cap.
Post-condició	Tenim les dades per a crear les parcel·les.

4.1.3 Cas d'ús per a ciutat sintètica

Aquest diagrama ens mostra els diferents paràmetres que pot triar l'usuari a l'hora de crear una ciutat sintètica, ja sigui amb un sol patró o amb diversos districtes amb diferents patrons.

Si tenim un sol patró, una vegada determinat aquest, caldrà que l'usuari pugui entrar algunes opcions per a cadascun d'ells, com el punt central i la freqüència dels carrers. Si tenim diferents districtes, l'usuari podrà escollir aquestes opcions per a cadascun dels patrons de cada districte.

L'usuari també podrà triar alguns paràmetres sobre la creació de carrerons (orientació dels carrers, amplada dels blocs), la divisió en parcel·les (amplada de les parcel·les) i el path on hi ha guardades les textures dels edificis.

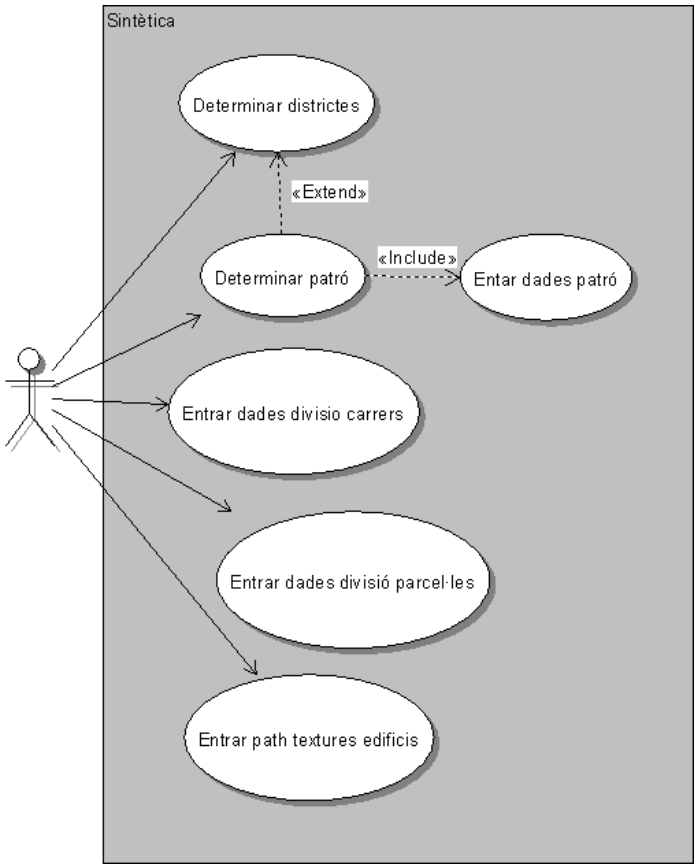


Figura 49: Diagrama de cas d'us per a ciutat sintètica

4.1.3.1 *Fitxes cas d'ús ciutat sintètica*

FITXA	Determinar districtes/patró
Funcionalitat	L'usuari entra les dades necessàries per a dibuixar el shader.
Actor(s)	Usuari.
Pre-condició	Cap.
Flux principal	1. L'usuari entra el path del mapa de districtes, si n'hi ha, i les relacions color/patró. 2. L'usuari entra per cada patró els paràmetres necessaris per a dibuixar-lo.
Subfluxos (exensions)	Cap.
Flux alternatiu	Cap.
Post-condició	Tenim les dades per a dibuixar el shader de la ciutat sintètica.

FITXA	Entrar dades divisió carrers
Funcionalitat	L'usuari entra les dades necessàries per a dividir les illes en carrers.
Actor(s)	Usuari.
Pre-condició	Cap.
Flux principal	1. L'usuari entra la rotació de les illes i la mida dels blocs.
Subfluxos (exensions)	Cap.
Flux alternatiu	Cap.
Post-condició	Tenim les dades per a crear els blocs.

4.1.4 Cas d'ús d'ajustament de paràmetres de control

Aquest diagrama mostra les possibilitats que té l'usuari per a canviar alguns paràmetres de la ciutat una vegada aquesta ja ha estat generada.

L'usuari podrà canviar la mida total de la ciutat, la mida i rotació de les illes, l'amplada de les parcel·les dintre les illes i la mida del pati interior. Aquests canvis tindran efecte en la ciutat resultant immediatament i podran ser canviats tantes vegades com l'usuari ho vulgui.

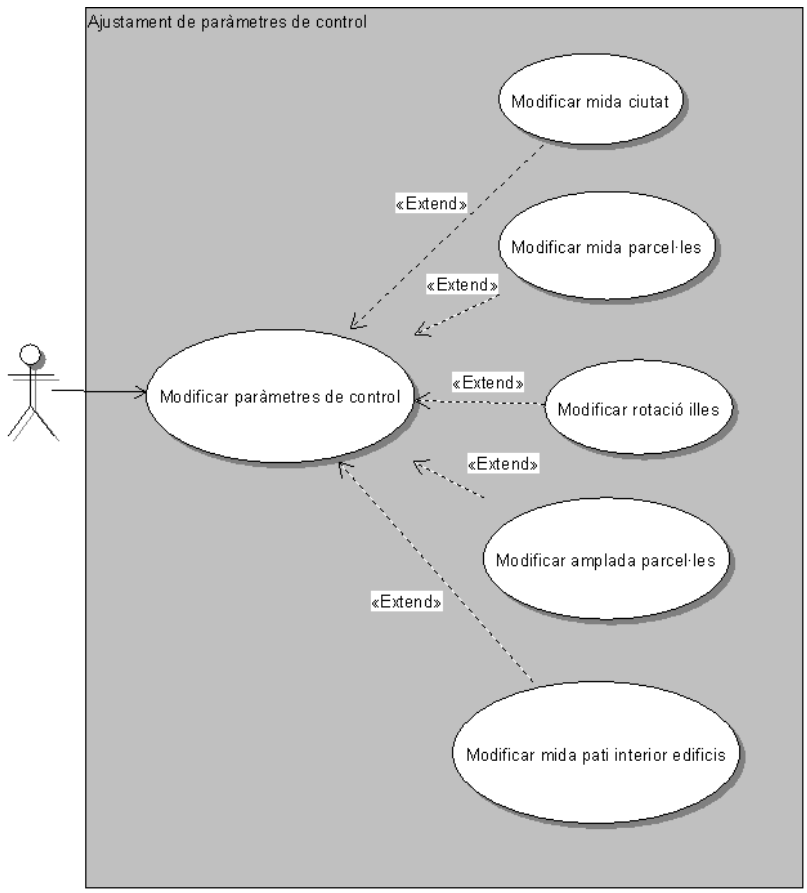


Figura 50: Diagrama de cas d'ús d'ajustament de paràmetres de control

4.1.4.1 Fitxes cas d'ús d'ajustament de paràmetres de control

FITXA	Modificar paràmetres de control
Funcionalitat	L'usuari fa modificacions a la ciutat ja generada.
Actor(s)	Usuari.
Pre-condició	Tenim una ciutat generada.
Flux principal	1. L'usuari pot modificar la mida de la ciutat, la mida i amplada de les parcel·les, el grau de rotació de les illes i la mida dels patis interiors dels edificis.
Subfluxos (exensions)	Cap.
Flux alternatiu	Cap.
Post-condició	La ciutat s'ha modificat segons els paràmetres que ha canviat l'usuari.

4.2 Diagrames d'activitat

Els diagrames d'activitat mostren fluxes de dades d'activitats i accions, amb suport per a decisions, iteracions i concurrència. En UML, els diagrames d'activitat es poden fer servir per descriure el negoci i fluxes de dades pas a pas en un sistema.

4.2.1 Diagrames d'activitat dels patrons grid i radial

Els diagrames d'activitat d'aquests dos patrons són molt semblants, ja que tant l'un com l'altre dibuixen dues col·leccions de línies en el pla per sumar-les al final. La diferència és que pel patró grid (Figura 51) dibuixem línies verticals i horitzontals per dibuixar una graella, mentre que amb el patró radial (Figura 52) dibuixem línies radials i angulars.

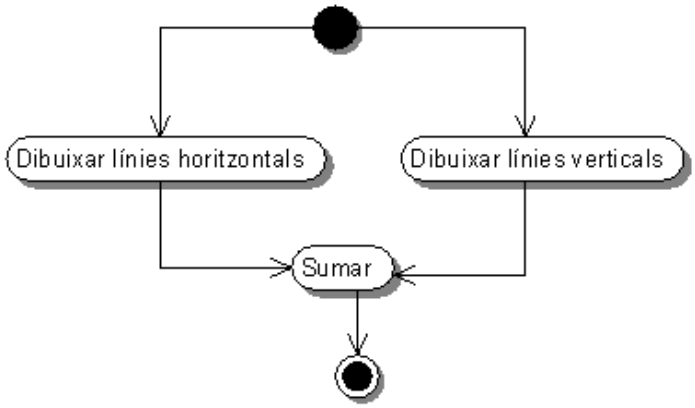


Figura 51: Diagrama d'activitat del patró grid

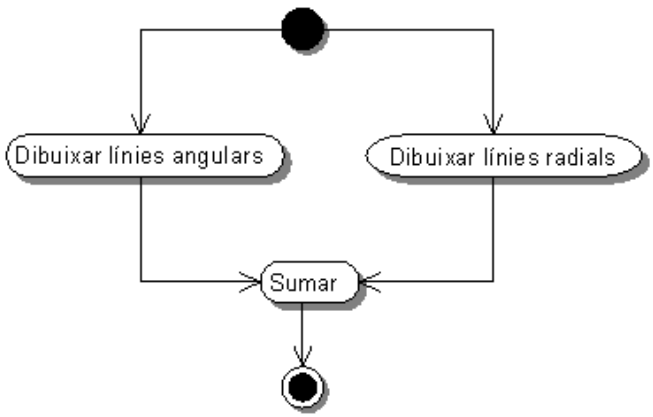


Figura 52: Diagrama d'activitat del patró radial

En podem veure un exemple de resultat a la Figura 53.

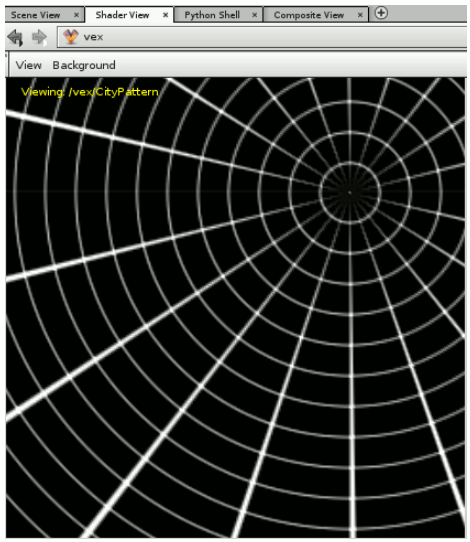


Figura 53: Exemple de patró radial generat

4.2.2 Diagrama d'activitat del patró voronoise

Per a calcular a quina cel·la de Voronoi pertany cada punt cal, per una banda, saber la seva posició respecte al centre i, per l'altra, obtenir resultat de la funció de densitat centrada (calculant la distància de cada punt amb el centre i aplicant-li una funció que assigni més densitat al centre que a la perifèria) o llegir un mapa de densitat, si en tenim. Amb aquestes dades, s'assignarà a cada punt un color indicant a quina cel·la pertany, de manera que al final haurem dibuixat un diagrama de Voronoi amb cada cel·la de color diferent.

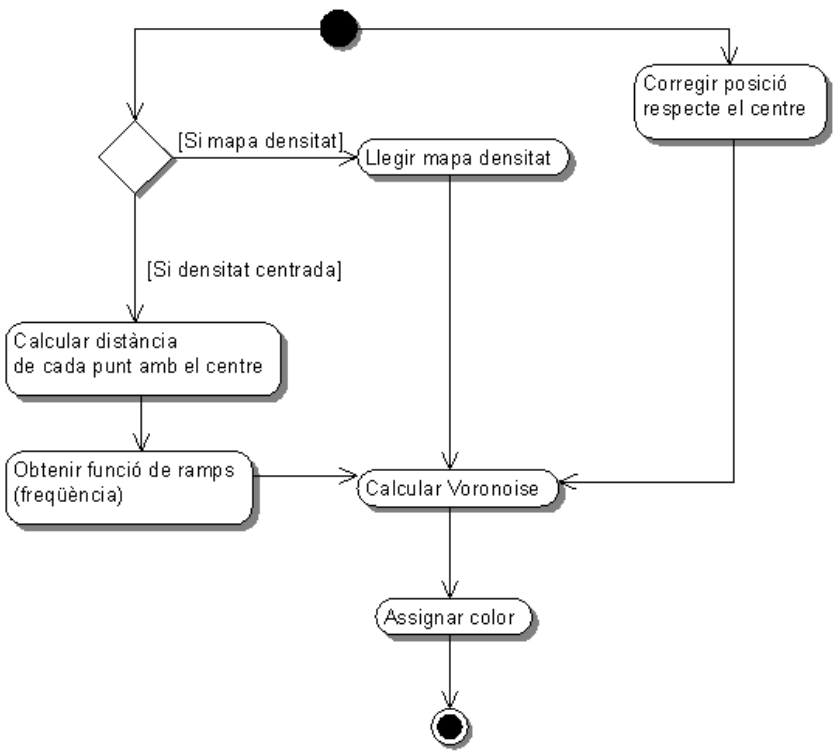


Figura 54: Diagrama d'activitat del patró voronoise

En podem veure en exemple de resultat a la Figura 55.

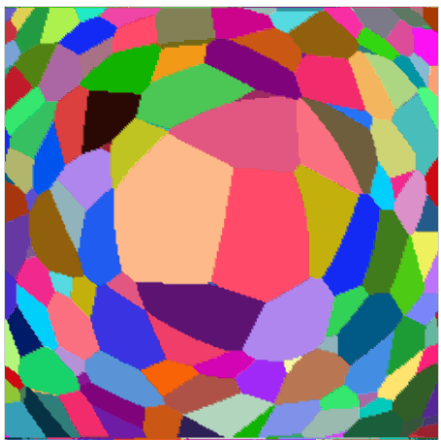


Figura 55: Exemple de patró voronoise generat

4.2.3 Diagrama d'activitat de ciutat amb districtes

En aquest cas, primer carreguem el mapa de districtes, que farà servir diferents colors per senyalitzar quins patrons corresponen a les diferents parts de la ciutat. Llavors, per cada color, mirarem si aquest és al mapa i, en cas afirmatiu, pintarem aquesta part del pla amb el patró corresponent (Figura 56).

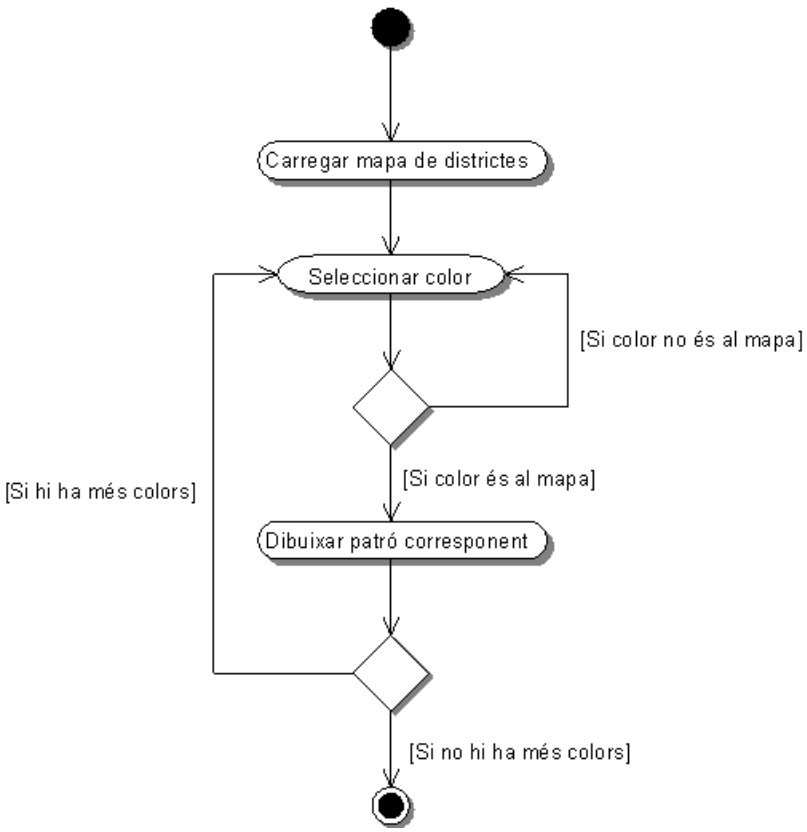


Figura 56: Diagrama d'activitat de ciutat amb districtes

4.2.4 Diagrama d'activitat de la xarxa COP2

En el cas de recreació de ciutats reals, no necessitem una xarxa COP2, ja que podem crear geometria directament del mapa que rebem com a entrada.

Per altra banda, en el cas de lectura des de la xarxa VEX, primer cal tractar la imatge de manera que sigui definida per crear-ne la geometria després. Per fer això, primer detectem els marges de la imatge i la binaritzem per eliminar la informació de color innecesària. Després n'eixamplarem les línes, de manera que siguin suficientment gruixudes per al processament posterior.

En cas de tenir mapa d'ús de la terra, l'escalem a la mateixa mida que la imatge de la xarxa VEX i la sobreposem, de manera que les zones que no siguin habitables quedin marcades com a tal. A la Figura 57 hi podem veure com quedaria un patró grid després de sobreposar-li un mapa d'ús de la terra.

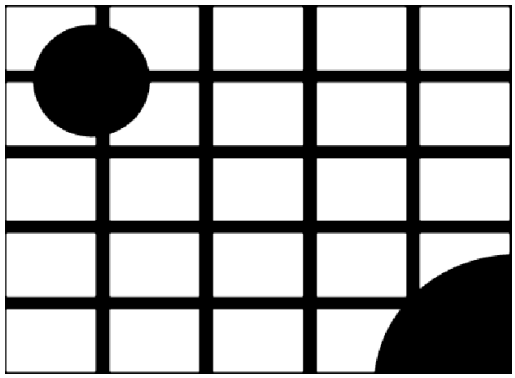


Figura 57: Imatge resultant d'un patró grid amb mapa d'ús de la terra

Si tenim un mapa de districtes, el binaritzem i tractem de la mateixa manera que hem fet amb la primera imatge, per així obtenir les avingudes que separen cada districte i sumar-ho al resultat obtingut del tractament de la imatge carregada de la xarxa VEX.

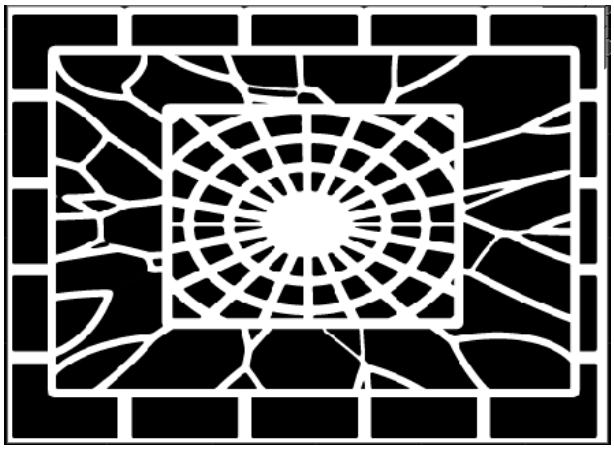


Figura 58: Exemple d'imatge COP2 amb districtes

A la Figura 59 podem veure el diagrama d'activitat de tot el procès.

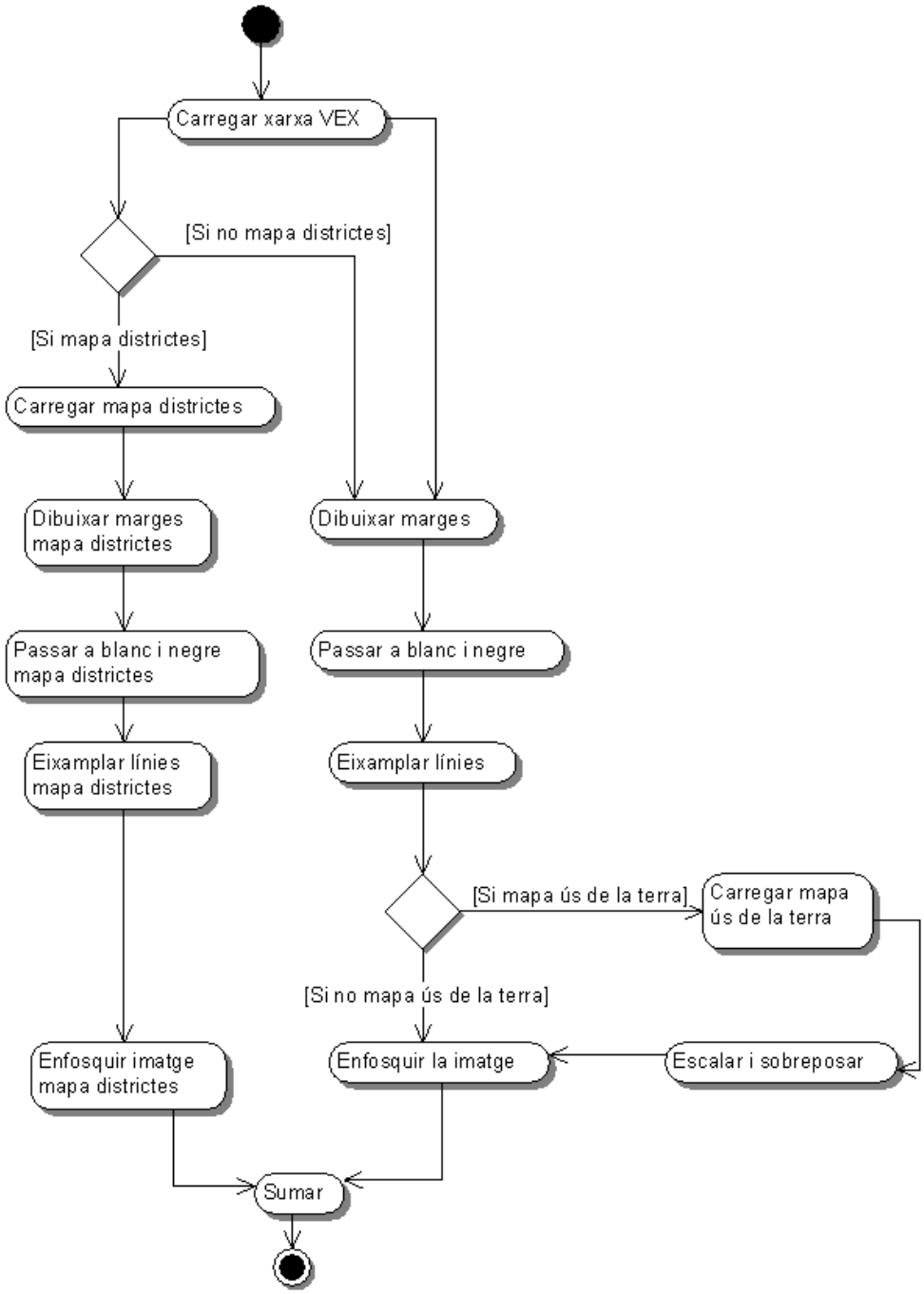


Figura 59: Diagrama d'activitat de la xarxa COP2

4.2.5 Diagrama d'activitat de crear carrers

Per a cada illa, creem una malla de caixes d'una mida i orientació determinades. Després l'intersequem amb la illa seleccionada per a obtenir els carrerons (Figura 60).

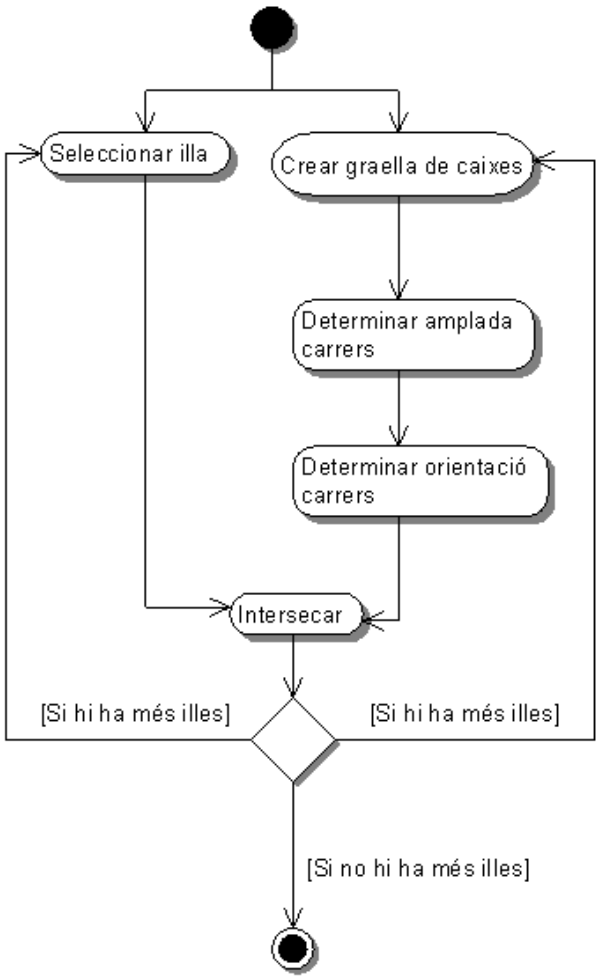


Figura 60: Diagrama d'activitat de crear carrers

Podem veure el resultat d'aquesta operació a la Figura 61.

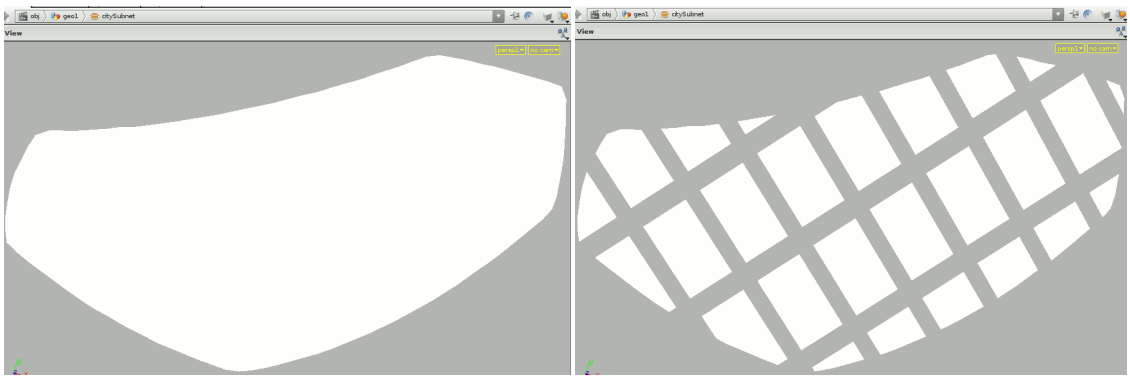


Figura 61: Una illa abans i després de la generació de carrers

4.2.6 Diagrama d'activitat de crear parcel·les

Primer de tot cal esborrar les parcel·les que siguin massa petites o massa grans per a la versemblança de la nostra ciutat. Una vegada fet això, seleccionem una parcel·la, li arrodonim els angles i creem una parcel·la interior (pati interior). Finalment, hi adaptem un edifici que s'hagi creat previament (Figura 62).

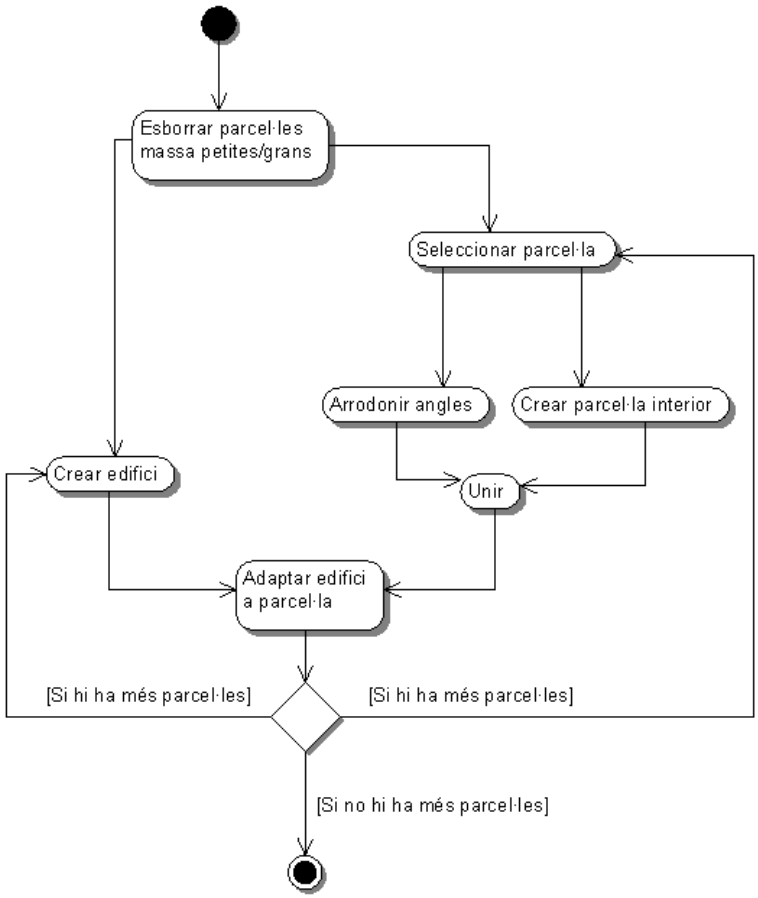


Figura 62: Diagrama d'activitat de crear parcel·les

Podem veure el resutat d'aquesta operació a la Figura 63.

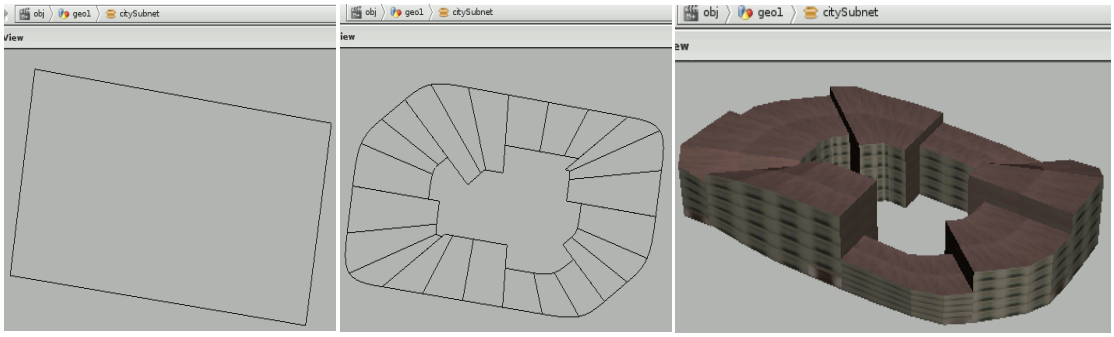


Figura 63: Exemple del procès de divisió en parcel·les

4.3 Diagrama de classes

Un diagrama de classes és un tipus de diagrama d'estructura estàtica que descriu l'estructura d'un sistema mostrant les classes del sistema, el seus atributs i les relacions entre elles.

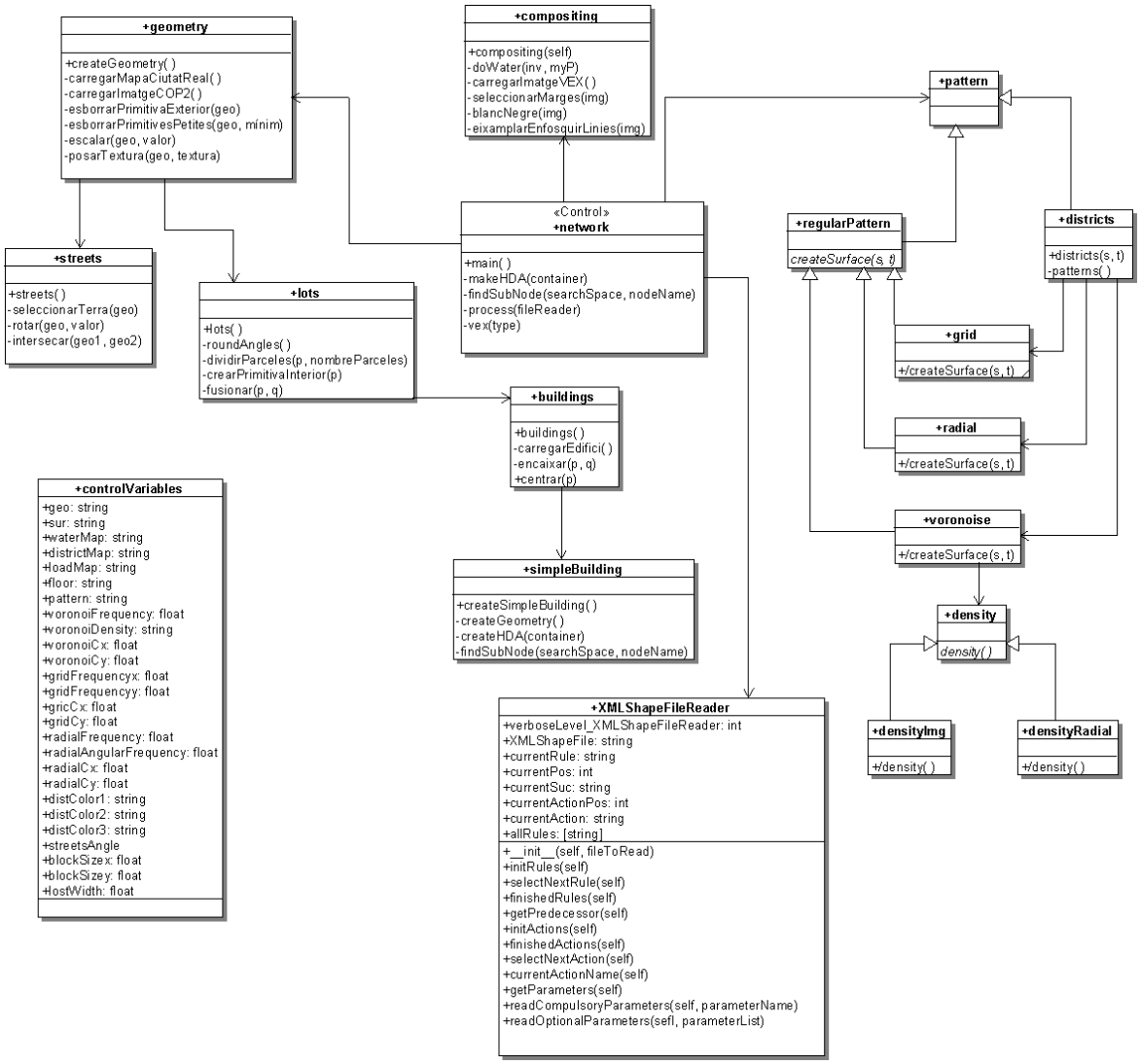


Figura 64: Diagrama de classes del sistema

A continuació es descriurà cadascuna de les classes que conformen el diagrama, indicant els seus mètodes:

4.3.1 Classe network

Aquesta classe actua com a classe de control, i s'encarrega de manejar el flux de dades per la resta de classes del sistema.

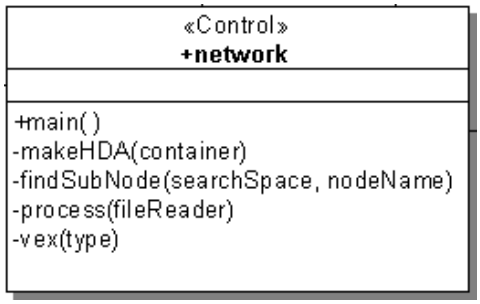


Figura 65: Classe network

- **main():** És el mètode principal, que s'encarrega de carregar el fitxer xml d'entrada i cridar els altres mètodes de la classe.
- **process(fileReader):** Identifica i obté tots els elements que prenen part en cadascuna de les regles, processant totes les accions seqüencialment. Per fer-ho, utilitzarem la classe XMLShapeFileReader (secció 4.3.15). Tot això es pot veure al codi de la Figura 66.

```
Process(fileReader):
      fileReader.initRules()
      while not fileReader.finishedRules():

            pre = fileReader.getPredecessor()

            fileReader.initActions()
            while not fileReader.finishedActions():

                  actionName = fileReader.currentActionName()

                  if actionName == 'districts': [...]
                  elif actionName == 'loadMap': [...]
                  elif actionName == 'landUse': [...]
                  elif actionName == 'voronoise': [...]
                  elif actionName == 'grid': [...]
                  elif actionName == 'radial': [...]
                  elif actionName == 'divideStreets': [...]
                  elif actionName == 'makeLots': [...]

                  fileReader.selectNextAction()

            fileReader.selectNextRule()
```

Figura 66: Codi per a identificar i processar regles

- **vex():** Mètode auxiliar de process(fileReader) que crea una xarxa VEX buida.
- **makeHDA(container):** Construeix un Houdini Digital Asset per a la geometria creada.
- **findSubNode(searchSpace, nodeName):** Mètode auxiliar de makeHDA(container) que retorna un node d'una xarxa indicant-li el seu nom.

4.3.2 Classe pattern

Classe abstracta de la que extendran els diferents patrons que puguem tenir.

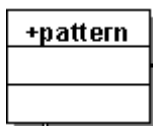


Figura 67: Classe pattern

4.3.3 Classe regularPattern

Classe abstracta que extén de pattern i de la que extendran les classes que representin els diferents patrons regulars.

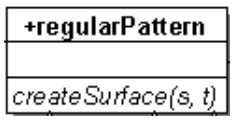


Figura 68: Classe regularPattern

- **createSurface():** Mètode abstracte que crea una xarxa VEX, on *s* i *t* representen les coordenades del punt actual.

4.3.4 Classe voronoise

Hem parlat del patró voronoise en la secció 3.2.1, i n'hem vist el diagrama d'activitat en la secció 4.2.2. Aquesta classe hereta de regularPattern.

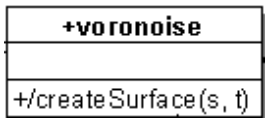


Figura 69: Classe voronoise

- **createSurface(s,t):** El node VORONOISE (secció 2.2.6) rep com a entrada un punt del pla i el resultat d'una funció de freqüència. Així doncs, per a cada punt de la textura en calculem la posició corregida, que és la seva posició respecte el centre (per al pseudocodi assumirem que el centre és el punt (0.5,0.5), però a la pràctica aquest punt podrà ser escollit per l'usuari). Per determinar la freqüència podem fer servir un mapa de densitat o bé calcular-la respecte el centre. En qualsevol dels dos casos, és la classe density qui se'n encarrega.

  El node VORONOISE ens retornarà quin és el punt de Vornoi més proper i li assignem un color en conseqüència, de manera que tots els punts que siguin més propers a aquell punt de Voronoi tinguin el mateix color, formant cel·les.

  A la Figura 70 hi veiem el codi que s'executa per a cada punt del pla.

```
createSurface(s,t):
      corregit = (s,t) – (0.5,0.5)
      freq = density()
      mésProper = VORONOISE(corregit, freq)
      valorFinal = Assignar color segons mésProper
      RETORNA valorFinal
```

Figura 70: Pseudocodi per a la generació d'un patró voronoise

Per a calcular la distància entre dos punts p_1 i p_2, el node DISTANCE fa servir la fòrmula que veiem a la Figura 71 , on x_1 i y_1 són les coordenades de p_1, i x_2 i y_2 ho són de p_2.

$$\text{dist}(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} = \sqrt{(p_2 - p_1) \cdot (p_2 - p_1)}$$

Figura 71: Fòrmula per a calcular la distància entre dos punts

4.3.5 Classe grid

Del patró grid n'hem parlat a la secció 3.2.2, i es pot trobar el diagrama d'activitat a la secció 4.2.1. Aquesta classe hereta de regularPattern.

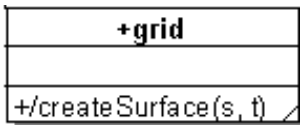


Figura 72: Classe grid

- **createSurface(s,t):** Com vam comentar a la secció 2.2.7, el node STRIPES retorna 0 o 1, determinant si en aquell punt "toca" pintar-hi una ratlla depenent de la freqüència que li passem per paràmetre. Així, per cada punt, s'evalua la funció de freqüència, tant horitzontalment com verticalment. Si en alguna d'aquestes dues evaluacions la funció retorna 1, aquell punt es pinta blanc. A la Figura 73 veiem el codi que s'executa per a cada punt (s,t) del pla.

```
createSurface(s,t):
      horitzontal = STRIPES(s,freq)
      vertical = STRIPES(t,freq)
      resultatFinal = horitzontal + vertical
      RETORNA resultatFinal
```

Figura 73: Pseudocodi per al patró grid

4.3.6 Classe radial

Del patró radial n'hem parlat a la secció 3.2.2, i es pot trobar el diagrama d'activitat a la secció 4.2.1. Aquesta classe hereta de regularPattern.

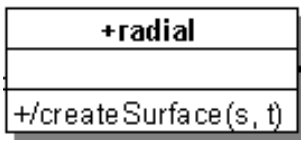


Figura 74: Classe radial

- **createSurface(s,t):** Per aconseguir aquest patró necessitarem dos nodes STRIPES (secció 2.2.7), un per a les línies radials i l'altres per a les línies angulars. Per a les línies radials, necessitem el radi, que es calcula determinant la distància del punt actual amb el centre amb la fòrmula de la Figura 71. Per a les línies angulars, necessitem l'arctangent de la divisió de les dues components del punt. Si una d'aquestes dues evaluacions retorna 1, es pinta aquell punt de blanc. Podem veure el codi que s'executa per a cada punt a la Figura 75.

```
createSurface(s,t):
      r = dist((s,t) – (0.5,0.5)) * multFreq
      radial = STRIPES(r)

      s = s – 0.5
      t = t – 0.5
      φ = tan⁻¹(s/t)
      angular = STRIPES(φ)

      valorFinal = radial + angular
      RETORNA valorFinal
```

Figura 75: Pseudocodi per al patró radial

4.3.7 Classe density i subclasses densityImg i densityRadial

En el cas del patró voronoise, en determina la densitat de les illes. Farem servir densityRadial quan la densitat hagi de ser radial, i densityImg quan l'haguem de calcular a partir d'un mapa de densitat.

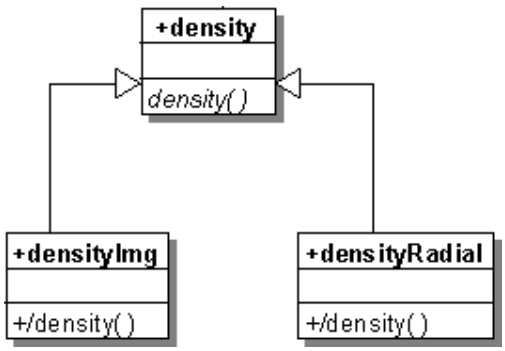


Figura 76: Classe density

- **density():** A partir d'un mapa de densitats (amb la classe densityImg), o calculant-la amb una funció respecte el centre (amb la classe densityRadial), crea el tros de xarxa VEX que determina la densitat. En el cas de fer servir la funció, necessitarem calcular la distància del punt actual corregit (com hem explicat a la secció 4.3.4) respecte el

centre utilitzant la fòrmula de la Figura 71. A la Figura 77 hi veiem el codi que s'executa en cas de tenir mapa de densitat, mentre que a la Figura 78 hi veiem el codi en el cas de voler densitat radial.

```
density():
      freq = lectura mapa densitat * multFreq
      RETORNA freq
```

Figura 77: Pseudocodi per al calcul de densitat amb mapa

```
density():
      freq = lectura mapa densitat * multFreq
      RETORNA freq
```

Figura 78: Pseudocodi per al càlcul de densitat radial

4.3.8 Classe districts

Extén de pattern i s'encarrega de crear una xarxa VEX per a una ciutat amb diversos patrons de districtes.

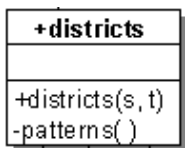


Figura 79: Classe districts

- **districts(s,t):** Interpreta el mapa de districtes, assignant un patró regular a cada color i creant-ne les subxarxes VEX corresponents.

  A vegades tindrem diferents patrons dintre una mateixa ciutat, tal i com està especificat a la secció 4.2.3. Per a aconseguir-ho, l'usuari haurà especificat un mapa de districtes i una llista de quins patrons corresponen a cada color del mapa. Per cada punt del mapa mirem quin color té i a quin patró correspon segons el que ens ha especificat l'usuari. A la Figura 80 hi veiem el codi que s'executa per a cada punt del pla.

```
districts(s,t):
      PER cada color c
            SI mapa(s,t) == c LLAVORS
                  dibuixar patró corresponent
            FSI
      FPER
      valorFinal = sumar patrons
      RETORNA valorFinal
```

Figura 80: Pseudocodi per a la separació en districtes

- **patterns():** Mètode auxiliar de districts() que crea una subxarxa VEX amb un patró regular específic.

4.3.9 Classe compositing

S'encarrega de crear la xarxa COP2 a partir d'una xarxa VEX.

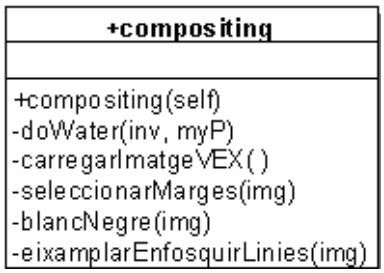


Figura 81: Classe compositing

- **compositing(self):** Una vegada obtingut el shader, caldrà tractar la imatge amb una xarxa COP2 per a ser utilitzada per a la creació de geometria, tal i com està explicat a la secció 4.2.4.

  Primer cal seleccionar només els marges de les figures de la imatge i passar-la a blanc i negre, ja que aquesta és la única informació que necessitarem. A continuació, hi sobreposem el mapa d'ús de la terra, si és que en tenim, amb la funció doWater. Finalment, eixamplem i enfosquim les línies per a què es vegin millor.

  En el cas que es tracti d'una ciutat amb districtes, cal afegir-hi carrers que els separin entre sí. Per aconseguir-ho, tractarem la imatge del mapa de districtes de la mateixa manera que ho hem fet amb la imatge del shader i, finalment, sobreposarem les dues imatges. A la Figura 82 hi podem veure el pseudocodi per tot aquest procés.

```
compositing():
      img = carregarImatgeVEX()
      self.seleccionarMarges(img)
      self.blancNegre(img)
      SI mapa ús de la terra LLAVORS
            self.doWater(img)
      FSI
      self.eixamplarEnfosquirLinies(img)
      SI mapa districtes LLAVORS
            mapaD = carregar mapa de districtes
            self.seleccionarMarges(mapaD)
            self.blancNegre(mapaD)
            self.eixamplarEnfosquirLinies(mapaD)
            img = img + mapaD
      FSI
      RETORNA img
```

Figura 82: Pseudocodi per al tractament d'imatges

- **doWater():** Si hi ha mapa d'ús de la terra, el carrega i el fusiona amb la imatge que estem tractant, que estarà en la variable img, com es veu al codi de la Figura 83.

```
doWater():
      mapaUsTerra = carregar mapa d'ús de la terra
      mapaUsTerra = escalar a mida de img
      mapaUsTerra = img + mapaUsTerra
      RETORNA mapaUsTerra
```

Figura 83: Pseudocodi per a intersecar el mapa d'ús de la terra

- **carregarImatgeVEX():** Carrega una imatge a partir d'un shader d'una xarxa VEX.
- **seleccionarMarges(img):** Esborra tots els elements de la imatge menys els contorns.

- **blancNegre(img):** Passa la imatge a blanc i negre.
- **eixamplarEnfosquirLinies(img):** Fa la imatge una mica borrosa per fer més amples les línies de la imatge i n'enfosqueix el resultat.

4.3.10 Classe geometry

Crea una geometria a partir d'una xarxa COP2 o d'un mapa d'una ciutat real, i la tracta fins a obtenir la nostra ciutat.

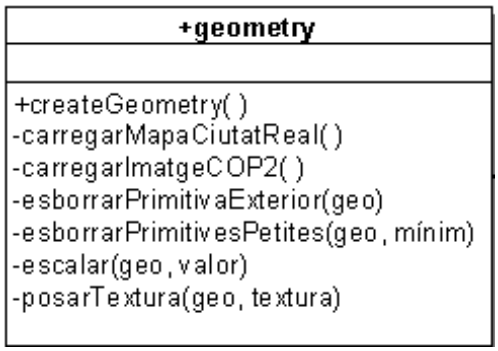


Figura 84: Classe geometry

- **createGeometry():** Crea la geometria bàsica, crida els diferents mòduls per crear-hi els carrerons i parcel·les, i en fa el tractament final.

  Per començar, carreguem la geometria bàsica a partir d'un mapa (si es tracta d'una ciutat real), o bé d'una imatge que haguem generat anteriorment (secció 6.3). En el segon cas, també cridem el mòdul per a la creació de carrers. A continuació esborrem les illes que hagin quedat massa petites, partint d'un mínim que haurà entrat l'usuari. Una vegada fet això, ja podem cridar el mòdul que subdividirà les illes en parcel·les. El resultat final podrà ser redimensionat en l'escala que esculli l'usuari. Addicionalment, si disposem de textura per al terra, creem un pla on l'enganxarem. Finalment, escalem el pla per a que tingui la mateixa mida que la nostra ciutat. Tot això es pot veure a la Figura 85.

```
createGeometry():
      SI ciutat real LLAVORS
            geo = carregarMapaCiutatReal()
      ALTRAMENT
            geo = carregarImatgeCOP2()
      FSI
      self.esborrarPrimitivaExterior(geo)
      SI NO ciutat real LLAVORS
            streets(geo)
      FSI
      esborrarPrimitivesPetites(geo, mínim)
      lots(geo,escala)
      self.escalar(geo)
      SI hi ha textura de terra LLAVORS
            terra = crearPla()
            self.posarTextura(terra, textura)
            self.escalar(terra,geo)
      FSI
```

Figura 85: Pseudocodi per a la generació de geometria

- **carregarMapaCiutatReal():** Crea una geometria a partir d'un mapa d'una ciutat.
- **carregarImatgeCOP2():** Crea una geometria a partir d'una imatge d'una xarxa COP2.
- **esborrarPrimitivaExterior(geo):** A vegades es crearà una primitiva que envolta la ciutat que no correspon a cap illa. Aquest mètode l'esborra.
- **esborrarPrimitivesPetites(geo, mínim):** Esborra les primitives més petites que el percentatge passat per paràmetre.
- **escalar(geo, valor):** Escala la geometria en cert valor o bé a la mida d'una altra geometria.
- **posarTextura(geo, textura):** Posa una textura a la geometria.

4.3.11 Classe streets

Divideix la geometria en carrerons.

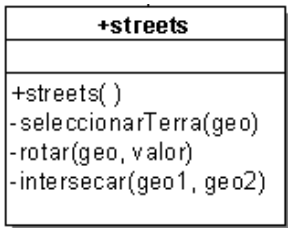


Figura 86: Classe streets

- **streets():** Aquest mòdul només serà cridat en el cas de creació de ciutats sintètiques, ja que en un mapa d'una ciutat real ja tindrem marcats els carrers. Aquest procés està especificat a la secció 4.2.5.

  Per a cada primitiva, cal calcular el nombre de divisions verticals i horitzontals. Per fer-ho, dividim l'àrea de la primitiva per l'àrea mitjana de totes les primitives i ho multipliquem pel nombre mitjà de divisions que ens agradaria tenir. D'aquesta manera, les primitives més grans tindran proporcionalment més divisions que les més petites.

  A continuació creem una caixa amb els nombres de divisions obtinguts. D'aquesta caixa ho esborrem tot menys el terra i n'obtenim un pla amb una sèrie de divisions, que conformaran els carrers de la illa. Després el fem rotar un nombre de graus escollit per l'usuari o bé aleatòriament. Finalment, ho intesequem amb la primitiva actual, que quedarà "tallada" en forma de carrers. A la Figura 87 hi veiem el codi que s'executa per a cada primitiva.

```
streets():
      divisionsX = midaPrimitiva / midaMitjana *
            nombreMitjaDivisionsX
      divisionsY = midaPrimitiva / midaMitjana *
            nombreMitjaDivisionsY
      caixa = crearCaixa(divisionsX,divisionsY)
      self.seleccionarTerra(caixa)
      self.rotar(caixa,valor)
      self.intersecar(caixa,p)
      RETORNA p
```

Figura 87: Pseudocodi per a la divisió en carrers

- **seleccionarTerra(geo):** Esborra tots els punts de la geometria menys el terra.
- **rotar(geo, valor):** Fa rotar la geometria un nombre determinat de graus.
- **intersecar(geo1, geo2):** Fa la intersecció de dues geometries.

A la Figura 88 veiem un exemple de com queda una primitiva després d'aplicar-li la operació de divisió en carrers.

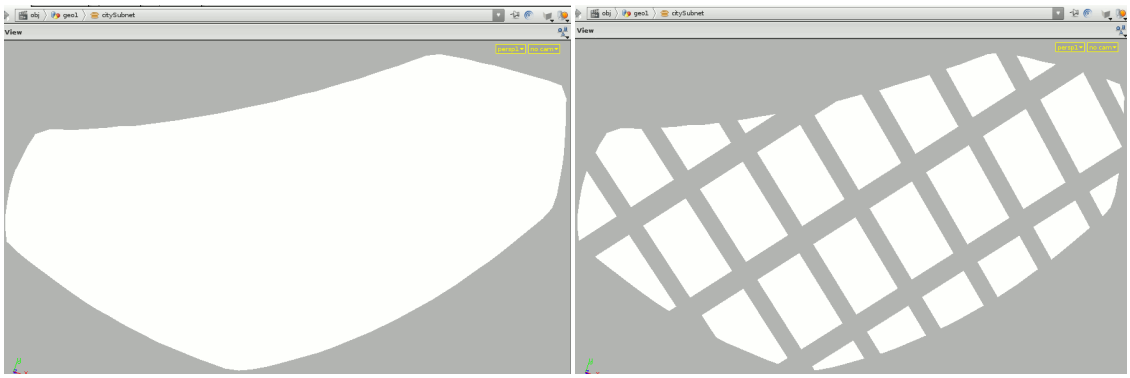


Figura 88: Una illa abans i després de la generació de carrers

4.3.12 Classe lots

Divideix els diferents blocs en parcel·les on després hi aixecarem els edificis.

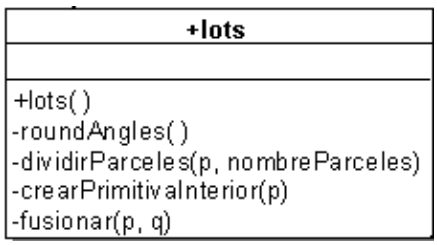


Figura 89: Classe lots

- **lots():** Aquest mòdul s'encarrega de subdividir cada illa en diverses parcel·les i a aixecar-hi els edificis, tal i com està especificat a la secció 4.2.6.

  Per tot això, primer arrodonim les cantonades de cada illa amb el mètode roundAngles per donar un aspecte menys quadriculat. Després, per calcular el nombre de parcel·les, multipliquem el perímetre de l'illa per la mitjana de perímetres de totes les illes i ho dividim pel nombre mitjà de parcel·les que ens agradaria tenir. D'aquesta manera, cada illa tindrà un nombre de parcel·les proporcional al seu perímetre.

  Dividim la illa en tants segments com parcel·les vulguem. Per cada un d'aquests segments creem un altre segment igual però més petit a l'interior de la primitiva, de

manera que quan els unim tots quedarà un pati interior. Finalment, cridem el mòdul d'edificis. A la Figura 90 podem veure el codi que executa tot aquest procés.

```
lots():

      self.roundAngles(p)

      nombreParceles = perimetrePrimitiva / perimetreMitjà *
      nombreMitjaParceles

      self.dividirParceles(p,nombreParceles)

      PER cada segment q FER

            interior = crearPrimitivaInterior(q)

            self.fusionar(q,interior)

            buildings(q)

      FPER
```

Figura 90: Pseudocodi per a la divisió en parcel·les

- **roundAngles(p):** Mètode auxiliar que arrodoneix els angles dels blocs .
- **dividirParceles(p, nombreParceles):** Divideix la illa p en el nombre de parcel·les passat per paràmetre.
- **crearPrimitivaInterior(p):** Crea una primitiva igual que p però més petita, que actuarà com a pati interior.
- **fusionar(p,q):** Fusiona dues geometries.

A la Figura 91 veiem un exemple de com queda una illa després d'aplicar-li la operació de divisió en parcel·les.

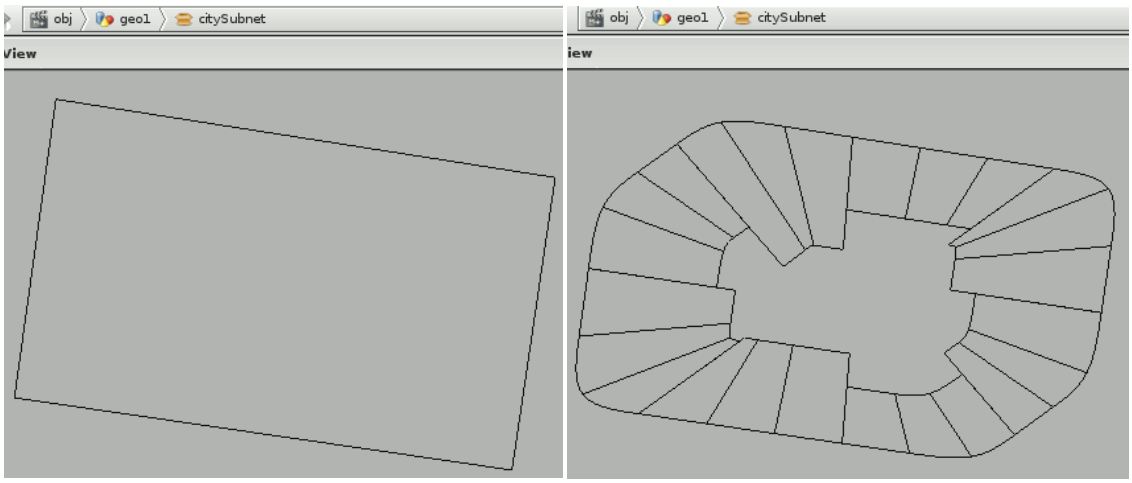


Figura 91: Una illa abans i després d'aplicar-li la divisió en parcel·les

4.3.13 Classe buildings

Posa els edificis a la nostra ciutat.

Figura 92: Classe buildings

- **buildings():** Agafa un model d'edifici i l'encaixa en cadascuna de les parcel·les de la nostra ciutat, amb altura variable. Primer carreguem un edifici que tinguem creat. A continuació l'encaixem en la parcel·la actual i l'acabem de centrar bé (Figura 93).

```
buildings():
      edifici = carregarEdifici()
      self.encaixar(edifici,p)
      self.centrar(edifici)
```

Figura 93: Pseudocodi per a la càrrega d'edificis

- **carregarEdifici():** Carrega la geometria d'un edifici que tinguem creat.
- **encaixar(p, q):** Fa encaixar la geometria p en el pla q.

- **centrar(p):** Aquest mètode serveix per assegurar que l'edifici queda ben centrat en el seu tros de pla corresponent.

A la Figura 94 veiem un exemple d'una illa una vegada hem aixecat edificis en totes les seves parcel·les.

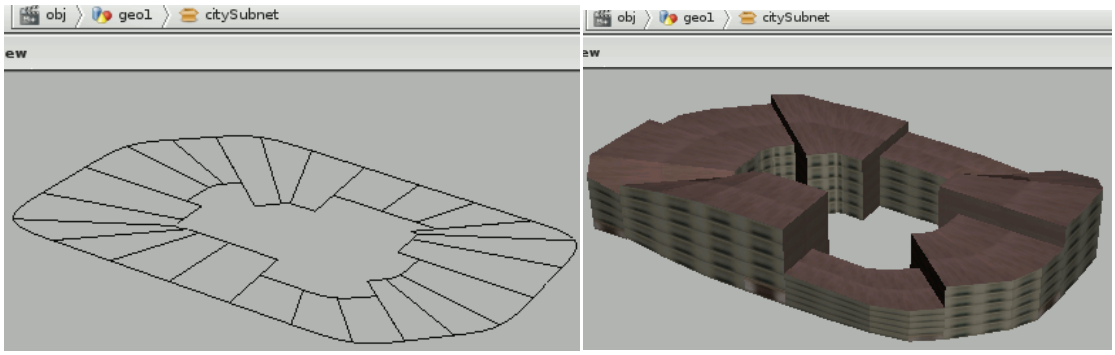


Figura 94: Illa abans i després d'insertar-hi edificis

4.3.14 Classe controlVariables

Conté les variables de control necessàries per la interacció entre classes.

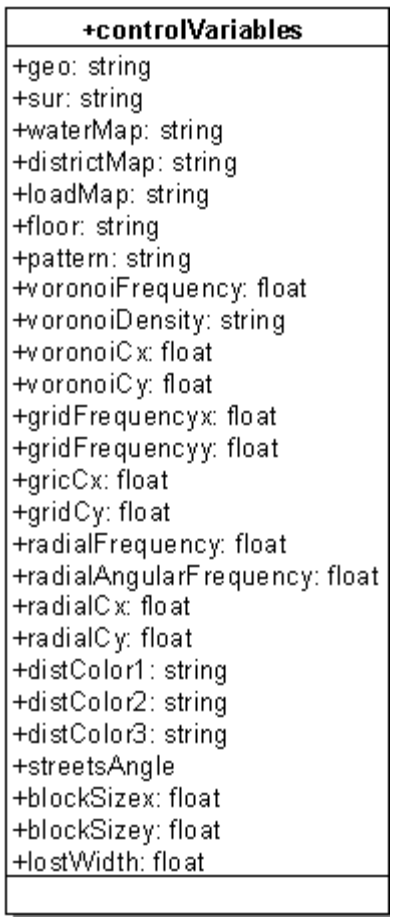


Figura 95: Classe controlVariables

4.3.15 Classe XMLShapeFileReader

Llegeix i processa les regles dels fixters XMLShape.

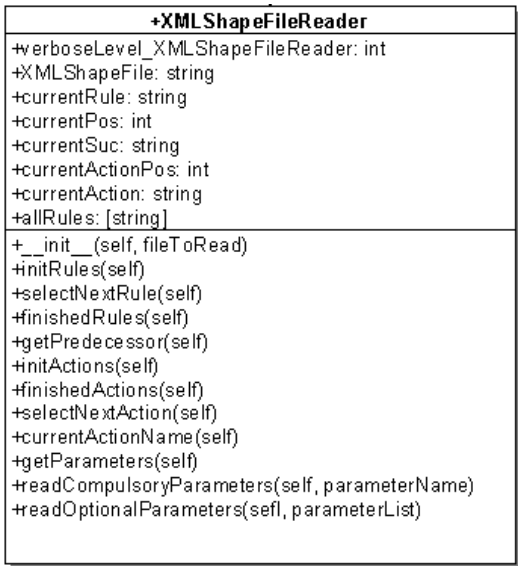


Figura 96: Classe XMLShapeFileReader

- **__init__(self, fileToRead):** Obre el fitxer XML i inicialitza paràmetres
- **initRules(self):** Carrega totes les regles i selecciona la primera.
- **selectNextRule(self):** Selecciona la regla següent.
- **finishedRules(self):** Comprova si hi ha més regles.
- **getPredecessor(self):** Retorna el predecessor de la regla actual.
- **initActions(self):** Carrega totes les accions i selecciona la primera.
- **selectNextAction(self):** Selecciona l'acció següent.
- **finishedAction(self):** Comprova si hi ha més accions.
- **currentActionName(self):** Retorna el nom de l'acció actual.
- **getParameters(self):** Retorna una llista amb tots els paràmetres de l'acció actual.
- **realCompulsoryParameters(self, parameterName):** Llegeix el valor dels paràmetres continguts a la llista parameterName, que seran obligatoris.
- **readOptionalParameters(self, parameterList):** Llegeix els valors dels paràmetres continguts a parameterList, que contindrà una llista de tuples de noms de node i valors per defecte.

4.3.16 Classe simpleBuilding

Crea la geometria d'un edifici simple.

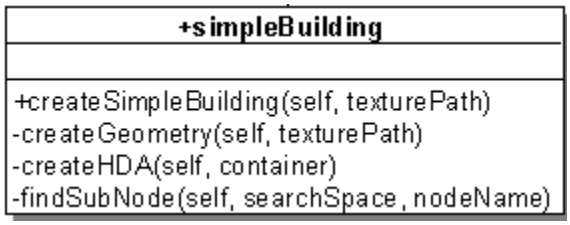


Figura 97: Classe simpleBuilding

- **createSimpleBuilding(self, texturePath):** Mètode principal que crida createGeometry i createHDA.
- **createGeometry(self, texturePath):** Crea la geometria de l'edifici i hi afegeix les textures.
- **createHDA(self, container):** Crea un Houdini Digital Asset per modificar alguns paràmetres de l'edifici.
- **findSubNode(self, serachSpace, nodeName):** Mètode auxiliar de makeHDA que retorna un node d'una xarxa indicant-li el seu nom.

4.4 Diagrames de seqüència

Un diagrama de seqüència és un tipus de diagrama d'interacció que mostra com els processos operen els uns amb els altres i en quin ordre.

Un diagrama de seqüència mostra amb línies paral·leles verticals (línies de vida) els diferents processos o objectes que viuen simultàniament i, amb fletxes horitzontals, els missatges intercanviats entre ells en l'ordre corresponent.

4.4.1 Diagrama de seqüència de recrear ciutats reals

Una de les dues operacions bàsiques de les que disposarem és la de recrear ciutats reals. Com es pot observar a la Figura 98, el mòdul *streetNetwork* recull els paràmetres que hagi entrat l'usuari i actua com a classe de control, coordinant els events necessàris per a dur a terme el nostre propòsit.

Primer de tot crida el mòdul *geometry,* que és el que comença a construir la geometria bàsica a partir del mapa que li haguem especificat a l'entrada, i s'encarrega de cridar el mòdul *lots* (per a la subdivisió dels blocs), que al seu torn crida el mòdul *buildings* (per a l'aixecament d'edificis).

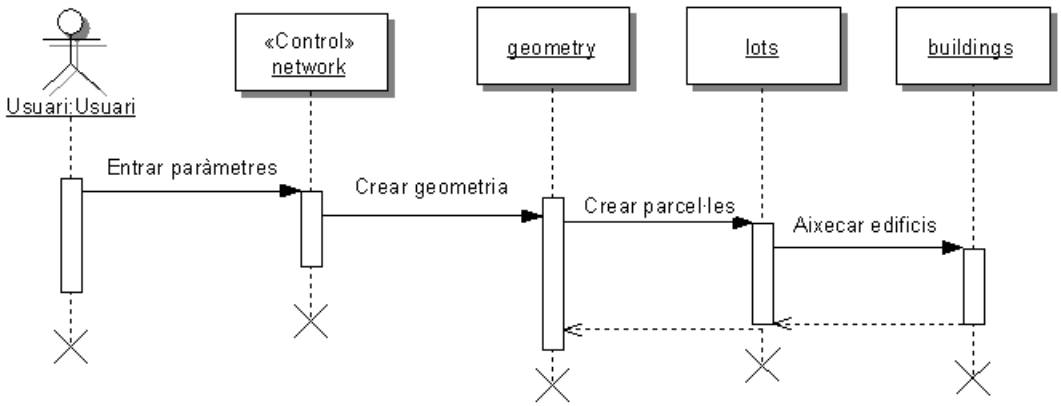


Figura 98: Diagrama de seqüència de recrear ciutats reals

4.4.2 Diagrama de seqüència de la creació d'una ciutat sintètica

L'altra operació bàsica és la de crear ciutats sintètiques. Dintre d'aquesta operació tindrem diferents possibilitats: fer tota la ciutat seguint un determinat patró o bé delimitar diversos districtes amb el seu propi patró. La Figura 99 mostra el diagrama de seqüència de crear una ciutat amb districtes.

De la mateixa manera que en el cas anterior, *streetNetwork* recull els paràmetres d'entrada i actua com a classe de control. Primer crida el mòdul *districts*, que coordina les classes

necessàries per a crear la xarxa VEX segons la distribució de districtes que li haguem especificat. Després crida *compositing* per a crear la xarxa COP2 que genera una imatge 2D a partir de la xarxa VEX anteriorment creada. Finalment, crida el mòdul *geometry*, que comença a crear la geometria bàsica de la ciutat i coordina els mòduls *streets* (per a la subdivisió en carrerons) i *lots* (per la subdivisió en blocs), que crida *buildings* per a aixecar-hi edificis.

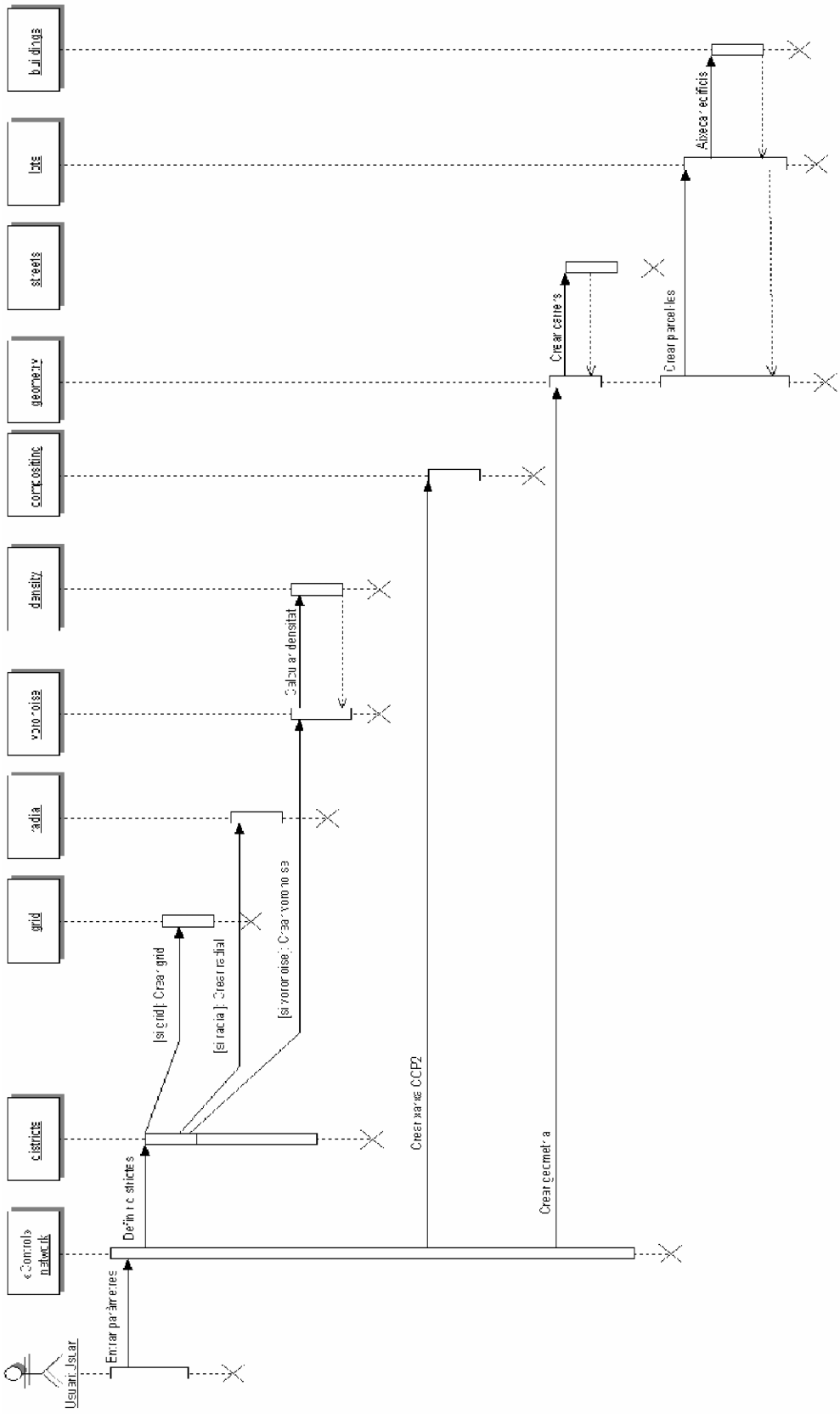


Figura 99: Diagrama de seqüència de crear ciutats sintètiques

Capítol 5: Disseny

En aquest capítol parlarem del disseny i sintaxi del XMLShape, llenguatge que farem servir per a llegir les regles que marcaran els paràmetres de les ciutats que generarem.

5.1 Disseny d'un CGA shape propi: XMLShape

Com s'ha dit al capítol anterior, el funcionament de projecte es basa en la sintaxi creada per Pascal Müller i Peter Wonka [5]. Tot i això, aquesta gramàtica s'ha hagut d'adaptar a conseqüència de la utilització d'un programa de modelatge 3D ja existent, amb les seves característiques particulars.

Aquesta gramàtica resultant, que a partir d'ara anomenarem XMLShape, funciona bàsicament igual que la seva antecessora, exceptuant algunes funcionalitats que no ha calgut dissenyar ja que el mateix Houdini les proporcionava.

Conceptualment, la sintaxi d'ambdós sistemes és similar[2]:

Id: predecessor → *successor*

D'aquesta manera s'obtenen el conjunt de regles necessàries per a dissenyar les xarxes de carrers.

A continuació s'explicarà el funcionament de les regles de l'XML shape.

[2] Per motius pràctics, en aquesta versió de l'XMLShape, s'ha simplificat la gramática excloent les condicions i el coeficient de probabilitat.

5.2 Sintaxi XML

Com s'ha comentat a la secció 2.9, el llenguatge escollit per a definir les ciutats és XML. S'ha creat una sintaxi basada conceptualment amb el CGA shape de Müller i Wonka i s'ha adaptat seguint l'estàndard que marca XML.

Les regles en general queden escrites de la següent manera:

```
<XMLshape>
      <header type="tipus" />
      <rule id="1">
            <predecessor name="pred"></predecessor>
            <successor>
                  <action name="act" params="num">
                        <param name="p1" value="x"/>
                        <param name="p2" value="y"/>
                              ...
                        <product name="prod"/>
                              ...
                  </action>
            </successor>
      </rule>
      ...
</XMLshape>
```

Seguint aquesta especificació, s'han de crear totes les regles necessàries per a generar la geometria que calgui.

5.2.1 Tags XML

<XMLshape> és el tag contenidor de tots els objectes. Serveix per a marcar l'inici i el final del fitxer XML, com també per identificar el tipus de fitxer que és.

```
<XMLshape>
      ...
</XMLshape>
```

<header> serveix per definir el tipus de projecte que estem definint. En el cas d'aquest projecte, sempre serà "XMLShape:streetNetwork".

<rule> defineix cadascuna de les regles del fitxer. Totes les regles estan identificades i són executades de manera seqüencial.

```
<rule id=”n”>

      ...
</rule>
```

<predecessor> és el tag que representa l'objecte al qual es vol aplicar l'acció descrita dins la regla. A part de la primera regla, que no s'aplica sobre cap objecte, ja que es crea geometria de nou, totes les altres s'han d'aplicar a un objecte predecessor determinat.

```
<predecessor name=”nom”></predecessor>
```

S'ha definit aquest tag amb un inici i un tancament pensant en la possibilitat, en un futur, d'afegir-hi paràmetres i condicions a dins del tag.

<successor> empaqueta totes les accions i paràmetres que es faran servir per a aplicar l'acció sobre el predecessor.

```
<successor>
      ...
</successor>
```

<action> és l'acció que s'ha d'aplicar sobre el predecessor. Segons el tipus d'acció descrit, s'executarà un mètode o un altre. El nombre de paràmetres que conté l'acció ve definit per l'atribut *parms*.

```
<action name=”nom” parms=”num”>

      ...
</action>
```

Es pot aplicar més d'una *action* sobre un mateix predecessor al definir-les totes dins d'un únic *successor*.

<param> són els tags que representen els paràmetres necessaris per a dur a terme l'acció requerida. Poden ser mides, coordenades, definicions de tipus, etc.

```
<param name=”nom” value=”valor”/>
```

<product> és el resultat d'aplicar l'acció a sobre el predecessor. Representa l'objecte resultant de les transformacions fetes per l'acció.

```
<product name=”nom”/>
```

Ala Figura 100 hi podem veure una representació d'aquests tags i quines relacions tenen entre ells.

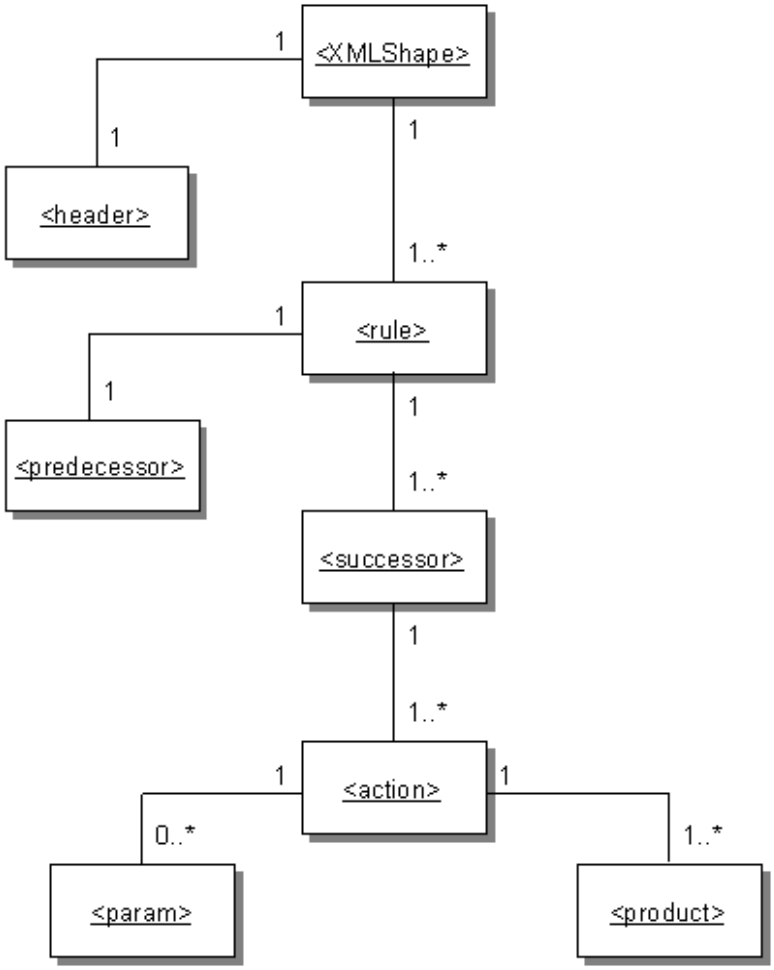


Figura 100: Representació conceptual l'estructura d'un fitxer XMLShape

5.2.2 Parser d'XML i DOM

Per a poder crear una estructura de dades a partir del fitxer XML cal utilitzar el parser que porta incorporat el Python. Exitsteix un mòdul anomenat Document Object Model (DOM) que conté totes les eines necessàries per a tractar els fitxers XML i recollir-ne les dades. En aquest cas s'ha utilitzat una simplificació d'aquest mòdul, anomenat *minidom*.

Un cop parsejat el fitxer, s'obté una estructura de dades en forma d'arbre, composta per nodes que segueixen l'especificació marcada per l'estàndard del DOM.

Utilitzant els mètodes que ens proporciona el *minidom* es pot navegar a través de tots els nodes i desar les dades que es considerin necessàries.

5.3 Un exemple

A continuació es mostrarà, pas per pas, la construcció d'una ciutat de patró radial.

Primer de tot, tal i com es veu a la Figura 101, cal especificar que estem utilitzant la sintaxi XMLshape i que es tracta d'un projecte del tipus streetNetwork:

```
<XMLshape>
      <header type="XMLShape:streetNetwork" />
            ...
</XMLshape>
```

Figura 101: Especificació del tipus de projecte

A continuació, dins d'aquests marcadors, ja es poden començar a especificar les regles. En la Figura 102 es veu com s comença definint la primera regla, on diem quin patró fem servir (en aquest cas radial) i quins paràmetres tindrà aquest patró.

```
      <rule id="1">
            <predecessor name="none" />
            <successor>
                  <action name="radial" params="4">
                        <param name="freqRadial" value="15" />
                        <param name="freqAngular" value="3" />
                        <param name="cx" value="0.75" />
                        <param name="cy" value="0.75" />

                        <product name="ground" />
                  </action>
            </successor>
      </rule>
```

Figura 102: Exemple de regla 1

Aquesta regla ens genera al Houdini la xarxa de nodes VEX que veiem a la Figura 103.

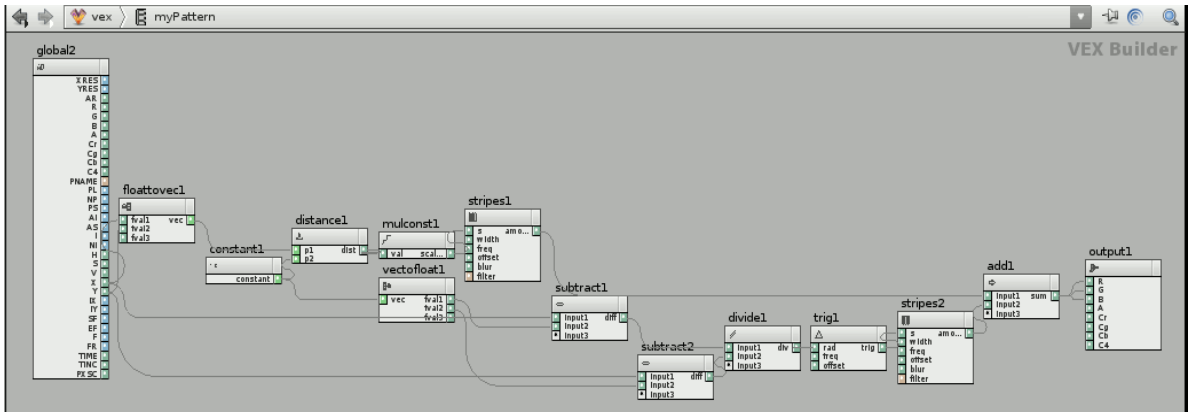


Figura 103: Xarxa VEX generada

El resultat que veiem a la vista Shader View és el que veiem a la Figura 104.

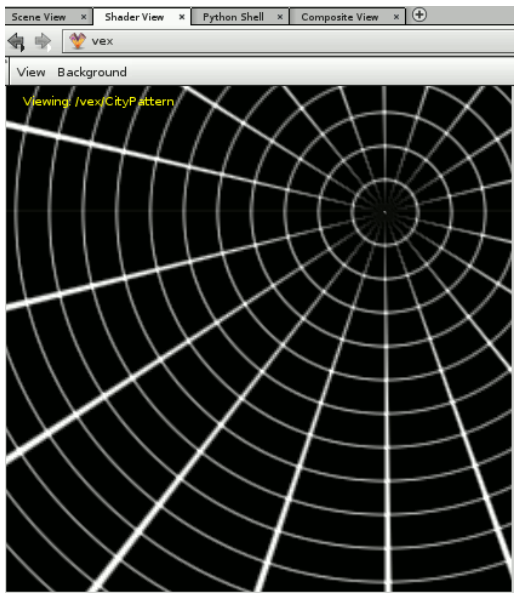


Figura 104: Resultat al Shader View

El següent pas és crear una imatge 2D a partir d'aquesta xarxa VEX i manipular-la fins a obtenir el resultat desitjat. Com veiem a la .

Figura 105, l'únic paràmetre que cal indicar és si farem servir un mapa d'ús de la terra.

```
<rule id="2">
      <predecessor name="ground" />
      <successor>
            <action name="landUse" params="1">
                  <param name="waterMap" value="None" />

                  <product name="streets" />
            </action>
      </successor>
</rule>
```

Figura 105: Exemple de regla 2

Això ens generarà la xarxa COP2 que veiem a la Figura 106.

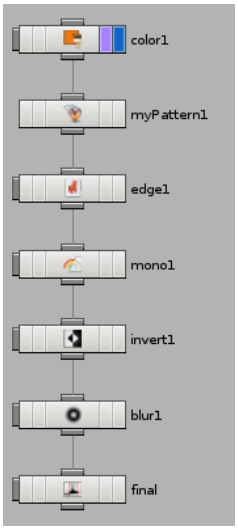


Figura 106: Xarxa COP2 generada

En la Figura 107 es mostra com a la vista Composite View podrem visualitzar la imatge generada per la xarxa de nodes.

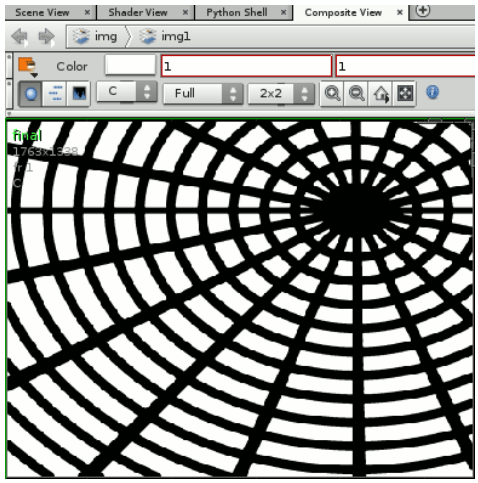


Figura 107: Resultat al Composite View

Una vegada obtinguda la imatge base, ja podem començar a construir la geometria 3D i fer-hi transformacions, la primera de les quals és crear els carrerons que subdivideixen les illes, com veiem a la Figura 108:

```
<rule id="3">
      <predecessor name="streets"/>
      <successor>
            <action name="divideStreets" params="5">
                  <param name="angle" value="random"/>
                  <param name="blockSizex" value="0.75"/>
                  <param name="blockSizey" value="0.8"/>

                  <product name="block"/>
            </action>
      </successor>
</rule>
```

Figura 108: Exemple de regla 3

A la Figura 109 veiem quin ha estat el resultat de crear els carrerons.

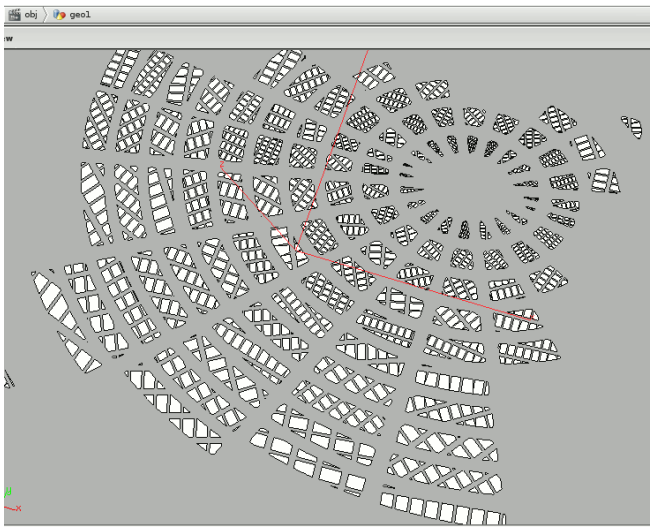


Figura 109: Resultat de la regla 3

A la Figura 110 veiem com la següent regla defineix com han de ser els blocs d'edificis que poblaran la ciutat.

```
<rule id="4">
      <predecessor name="block" />
      <successor>
      <action name="makeLots" params="1">
            <param name="width" value="0.31" />

            <product name="lot" />
          </action>
      </successor>
</rule>
```

Figura 110: Exemple de regla 4

Això generarà divisions, com veiem a laFigura 111, on més tard hi afegirem edificis.

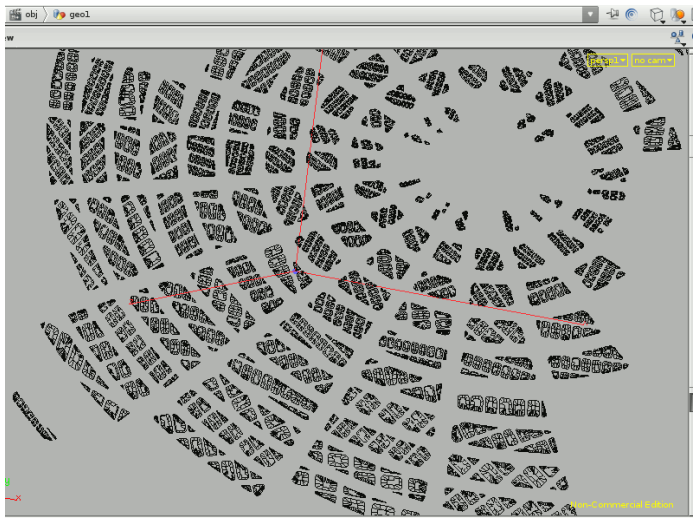


Figura 111: Resultat de la regla 4

Finalment, la última regla definirà en quin directori tenim guardades les imatges que farem servir per a les textures dels edificis, com veiem en la Figura 112.

```
<rule id="5">
            <predecessor name="lot" />
            <successor>
                  <action name="createBuildings" params="1">
                        <param name="texturePath" value="C:/Documents
and Settings/Mei R/My Documents/ETIG/PFC/Cities/" />

                        <product name="building" />
                  </action>
            </successor>
      </rule>
```

Figura 112: Exemple de regla 5

El resultat final de totes les regles és el que veiem a la Figura 113.

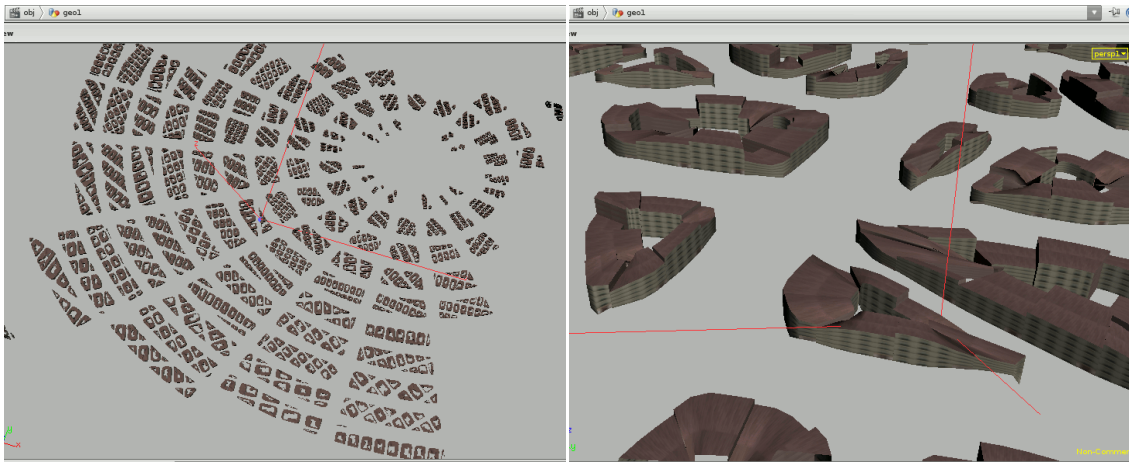


Figura 113: Resultat final

A la Figura 114 veiem l'arbre de nodes del DOM (tal i com està explicat a la secció 5.2.2) per al fitxer XMLShape fet servir en aquest exemple:

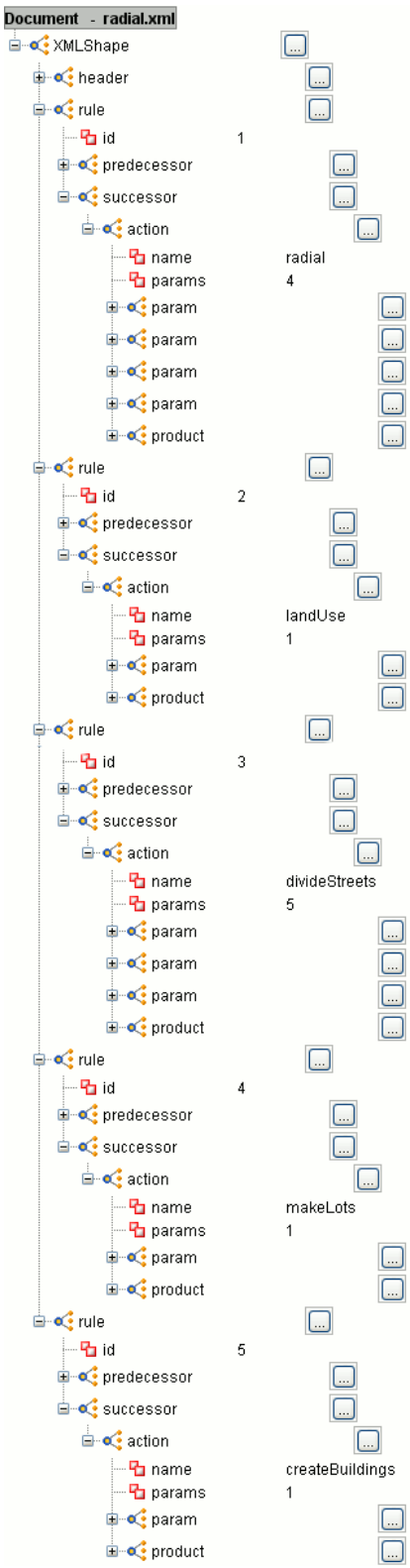


Figura 114: Arbre de nodes

De manera semblant tindrem regles per a definir altres patrons (grid i voronoise), distribució de districtes, llegir mapes de ciutats reals, etc.

Capítol 6: Implementació

Dins aquest capítol es mostrarà com s'ha implementat el disseny vist als capítols anteriors i com s'ha creat el codi necessari per a interpretar les regles del XMLshape. També es veurà la manera com les tres eines que conformen el projecte (Houdini, Python i XML) s'entrellacen per tal d'aconseguir el resultat desitjat.

6.1 Càrrega de l'XML

El primer pas a seguir és llegir el fitxer XML creat per l'usuari i carregar-lo a la memòria del Houdini per tal de poder generar la geometria seguint les regles marcades al fitxer. Per això es fa servir el mòdul *minidom* de Python, del que ja s'ha parlat a l'apartat 5.2.2.

Per a carregar el fitxer de regles, es fan servir les sentències mostrades a la Figura 115:

```
fitxer = open('fitxer.xml')
doc = minidom.parse(fitxer)
```

Figura 115: Codi per a carregar fitxer XML

Un cop executat, la variable *doc* conté un arbre de nodes amb totes les dades del fitxer (Figura 100). A partir d'aquí es van tractant les regles de manera seqüencial, accedint a elles amb els mètodes que proporciona el DOM.

6.2 Definició de patrons (VEX)

Per a crear un shader en una xarxa VEX (secció 2.2), primer caldrà crear un contenidor del tipus *COP2 Filter*, per després poder-lo fer servir en una xarxa COP2 sense problemes (Figura 116).

```
sur = hou.node('/vex').createNode('cop2filter')#create filter
sur.setName('myPattern') #change its name
```

Figura 116: Codi per a crear un contenidor VEX

A partir d'aquest moment ja podrem anar creant els nodes necessàris dintre d'aquest contendior.

6.2.1 Patró voronoise

Per poder definir el patró voronoise, primerament hem de definir una funció de densitat. Aquesta funció pot ser llegida d'una imatge, o calculada analíticament. A continuació,

explicarem el cas de densitat definida a una imatge, per després explicar el cas de la definició analítica amb una funció de densitat radial, per, finalment, explicar com s'implementa el patró de voronoise.

6.2.1.1 Funció de densitat definida a una imatge

Una opció que dóna un gran control a l'usuari és definir una funció de densitat mitjançant una imatge. Per carregar-la, el módul de densitats crea un node COLORMAP i li dóna com a paràmetre el camí al fitxer d'imatge. Després, això s'ha de conectar a les variables que proporciona Houdini per a les coordenades de la posició de textura que s'estan avaluant, i que tenim referenciades amb la variable *glo*. Finalment, el node a conectar a la sortida del VEX és el node *text*. Veure Figura 117.

```
text = sur.createNode('colormap')
text.parm('cmap').set(controlVariables.voronoiDensity)
text.setInput(1,glo,22)
text.setInput(2,glo,23)
```

Figura 117: Part del codi que recull un mapa de densitats

6.2.1.2 Funció de densitat radial

A continuació explicarem el codi necessari per generar una funció de densitat radial. En aquest codi, *sur* és una variable que respresenta la superfície VEX sobre la que estem generant el patró Voronoise. El primer a fer és agafar el node *constant1*, que conté la constant que conté la posició del centre de la funció, i en calculem la distància euclídea al punt actual amb l'expresió de la fòrmula de la Figura 71. Podem veure aquest codi a la Figura 118.

```
cons = sur.node('constant1')
#compute the distance between
#  the center point and the current point
dist = sur.createNode('distance')
dist.setFirstInput(vec)
dist.setInput(1,cons)
...
```

Figura 118: Part del codi per a la densitat radial

Després, invertim la funció amb un node COMPLEMENT, ja que volem que la freqüència augmenti en acostar-nos al centre i no al revés (Figura 119).

```
#make the frequency decreasing instead of increasing
comp = sur.createNode('complement')
comp.setFirstInput(dist)
...
```

Figura 119: Part del codi per a la densitat radial

A continuació, el node RAMPS ens permet tenir control sobre aquest decreixement de la freqüència (Figura 120).

```
#have control of the decrency of the frequency
ramp = sur.createNode('ramps')
ramp.setFirstInput(comp)
ramp.parm('freq').set(0.93)
...
```

Figura 120: Part del codi per a la densitat radial

Finalment, convertim aquest valor float a un vector i ho multipliquem per una constant per obtenir més cel·les (Figura 121).

```
fltovec2 = sur.createNode('floattovec')
fltovec2.setFirstInput(ramp)
fltovec2.setInput(1,ramp)
fltovec2.setInput(2,ramp)

#multiply the frequency by a constant
mult = sur.createNode('mulconst')
mult.parm('mulconst').set(controlVariables.voronoiFrequency)
```

Figura 121: Part del codi per a la densitat radial

6.2.1.3 *Definició del patró voronoise*

Per a generar el patró voronoise necessitarem un node VORONOISE (del que hem parlat a la secció 2.2.6). A aquest li passem la densitat calculada, representada per la variable *mult*, i la posició actual respecte el centre, representada per la variable *sub* (Figura 122).

```
vor = sur.createNode('voronoise')
vor.setInput(1,mult)
vor.setFirstInput(sub)
...
```

Figura 122: Part del codi per a definir el patró voronoise

A continuació, utilitzem un node RANDOM per a que ens retorni un color aleatrori segons el valor obtingut, que endollarem a la sortida de la variable *vor*, que representa el node VORONOISE (Figura 123).

```
#create a random node, link it to voronoi noise
ran = sur.createNode('random')
ran.parm("signature").set("vv")
ran.setFirstInput(vor,2)
```

Figura 123: Part del codi per a definir el patró voronoise

6.2.2 Patró grid

Per a la generació d'aquest patró, crearem dos nodes STRIPES (del que hem parlat a la secció 2.2.7), un per a les línies horitzontals i l'altre per a les línies verticals, passant-li al primer la posició sobre l'eix de les x i al segon la posició sobre l'eix de les y, que estan referenciades a la variable *glo* (Figura 124) .

```
#horizontal stripes
str1 = sur.createNode('stripes')
str1.parm('width').set(0.05)
str1.parm('freq').set(5)
str1.parm('blur').set(1)

#vertical stripes
str2 = sur.createNode('stripes')
str2.parm('width').set(0.05)
str2.parm('freq').set(5)
str2.parm('blur').set(1)

str1.setFirstInput(glo,22)
str2.setFirstInput(glo,23)
...
```

Figura 124: Part del codi per a la generació del patró grid

Finalment, sumarem els resultats d'aquests dos nodes (Figura 125).

```
add = sur.createNode('add')
add.setFirstInput(str2)
add.setInput(1,str1)
```

Figura 125: Part del codi per a la generació del patró grid

6.2.3 Patró radial

Per a la generació del patró radial, igual que amb el patró grid, necessitarem dos nodes STRIPES (explicat a la secció 2.2.7), però, en aquest cas, necessitarem afegir-hi les operacions descrites a la secció 4.3.6.

Primer crearem el node STRIPES corresponent a les línies radials, al que li passarem el radi (distància del punt actual respecte el centre, calculat amb la fòrmula de la Figura 71). Podem veure aquest codi a la Figura 126.

```
#compute the distance between the center point
#and the current point
dist = sur.createNode('distance')
dist.setFirstInput(fltovec)
dist.setInput(1,cons)

#the radial frequence
mulconst = sur.createNode('mulconst')
mulconst.setFirstInput(dist)
mulconst.parm('mulconst').set(15)
#the radial stripes
str = sur.createNode('stripes')
str.setFirstInput(mulconst)
str.parm('width').set(0.05)
str.parm('blur').set(1)
...
```

Figura 126: Part del codi per a la generació del patró radial

L'altre node STRIPES correspondrà a les línies angulars, que necessitarà l'angle com a entrada (Figura 127).

```
div = sur.createNode('divide')
div.setFirstInput(sub)
div.setInput(1,sub2)

trig = sur.createNode('trig')
trig.setFirstInput(div)
trig.parm('func').set('vop_atan') #arctangent

#the angular stripes
str2 = sur.createNode('stripes')
str2.setFirstInput(trig)
str2.parm('width').set(0.05)
str2.parm('freq').set(3)
str2.parm('blur').set(1)
```

Figura 127: Part del codi per a la generació del patró radial

6.2.4 Districtes

Per definir patrons segons diferents districtes (tal i com hem explicat a la secció 4.3.8), primer caldrà carregar el mapa de districtes i endollar-lo a la variable glo, que referencia la posició actual. Després convertim la sortida a float, per poder-ne evaluar les components (Figura 128).

```
mapfloat = sur.createNode('vectofloat')
map = sur.createNode('colormap')
map.parm('cmap').set(controlVariables.districtMap)
map.setInput(1,glo,22)
map.setInput(2,glo,23)
mapfloat.setFirstInput(map)
...
```

Figura 128: Part del codi per a la generació de districtes

A continuació, cal comprovar les relacions color-patró entrades en el fitxer XML. Així, per cada patró crearem un node CONSTANT (secció 2.2.3) que contindrà el color corresponent i el convertirem a float (Figura 129).

```
voronoiColor = sur.createNode('vectofloat')
if controlVariables.distColor1=='red':
      red = sur.createNode('constant')
      red.parm('consttype').set(16)
      red.parm('colordefr').set(1)
      voronoiColor.setFirstInput(red)
elif controlVariables.distColor1=='green':
      green = sur.createNode('constant')
      green.parm('consttype').set(16)
      green.parm('colordefb').set(1)
      voronoiColor.setFirstInput(green)
elif controlVariables.distColor1=='blue':
      blue = sur.createNode('constant')
      blue.parm('consttype').set(16)
      blue.parm('colordefg').set(1)
      voronoiColor.setFirstInput(blue)
...
```

Figura 129: Part del codi per a la generació de districtes

Després creem un node COMPARE (secció 2.2.4) per a comparar cada component del punt del mapa amb el color (Figura 130).

```
c11 = sur.createNode('compare')
c11.setFirstInput(voronoiColor)
c11.setInput(1,mapfloat)
c12 = sur.createNode('compare')
c12.setFirstInput(voronoiColor,1)
c12.setInput(1,mapfloat,1)
c13 = sur.createNode('compare')
c13.setFirstInput(voronoiColor,2)
c13.setInput(1,mapfloat,2)
...
```

Figura 130: Part del codi per a la generació de districtes

Finalment, fem una operació AND de les tres comparacions, i creem un node IF(secció 2.2.5), dintre del qual crearem la subxarxa per a la generació del patró corresponent, tasca de la que s'encarrega el mètode *patterns* (Figura 131).

```
voronoiand = sur.createNode('and')
voronoiand.setFirstInput(c11)
voronoiand.setInput(1,c12)
voronoiand.setInput(2,c13)

if1 = sur.createNode('if')
if1.setFirstInput(voronoiand)
self.patterns(type,if1.node('suboutput1'))
```

Figura 131: Part del codi per a la generació de districtes

Es repeteixen aquests passos per a cada patró implementat.

6.3 Tractament imatge (COP2)

Primer de tot cal crear un contenidor per a xarxes COP2 (secció 2.3) on hi crearem una imatge en blanc amb un node COLOR per enganxar-li la imatge del shader (que anteriorment hem anomenat *myPattern*, veure secció 6.2) . Podem veure aquesta part del codi a la Figura 132.

```
comp = hou.node('/img').createNode('img')

col = comp.createNode('color')
col.parm('overridesize').set(1)
col.parm('size1').set(1828)
col.parm('size2').set(1332)


myP = comp.createNode('myPattern')
myP.setFirstInput(col)
```

Figura 132: Part del codi per al tractament d'imatge

6.3.1 Mapa d'ús de la terra

Si s'ha introduit un mapa d'us de la terra al fitxer XML, cal carregar-lo amb un node FILE (explicat a la secció 2.3.1) i escalar-lo a la mateixa mida que la imatge del shader amb un node SCALE (explicat a la secció 2.3.4). Finalment, sobreposem les dues imatges amb un node MIN, que retorna el valor mínim de les imatges píxel a píxel, de manera que es queda amb les parts més fosques (Figura 133).

```
file = comp.createNode('file')
file.parm('filename1').set(controlVariables.waterMap)

scale = comp.createNode('scale')
scale.setFirstInput(file)
scale.setInput(1,myP)
scale.parm('scaletype').set(3)

min = comp.createNode('min')
min.setFirstInput(scale)
min.setInput(1,inv)
```

Figura 133: Part del codi per a sobreposar un mapa d'ús de la terra

6.4 Creació geometria (SOP)

Primer cal crear un contenidor per a una xarxa SOP (secció 2.4). Podem veure aquest codi a la Figura 134.

```
geo = hou.node('/obj').createNode('geo')
...
```

Figura 134: Part del codi per a la creació de geometria

Per a carregar la geometria bàsica, farem servir un node TRACE. Aquest crearà una geometria a partir d'un mapa si en tenim, o a partir de la imatge d'una xarxa COP2 si estem creant una ciutat sintètica (Figura 135).

```
tra = geo.createNode('trace')
if controlVariables.loadMap != None:
      tra.parm('file').set(controlVariables.loadMap)#use the map
      tra.parm('usecop1').set(0)
else:
      tra.parm('coppath').set('/img/img1/final')#use COP2
      tra.parm('usecop1').set(1)
tra.parm('doresample').set(1)
tra.parm('step').set(15)
```

Figura 135: Part del codi per a la creació de geometria

6.4.1 Divisió carrerons

Per a la creació de carrerons (operació descrita a la secció 4.3.11), primer crearem una malla regular de caixes, amb un cert nombre de divisons. Per a que aquest nombre de divisions sigui proporcional amb la mida de cada illa, farem servir l'expressió que veiem a la Figura 136. En aquesta, dividim l'àrea de la illa actual (calculada al node *measureLot*) per l'àrea mitjana (calculada al node *avgarea*) i ho multipliquem pel nombre de divisions mitjà que volem tenir. Finalment hi sumem 2 per a que el resultat mai sigui 0 o 1 (Figura 136).

```
box = geo.createNode('box')
box.parm('type').set(1) #change the type to polygon mesh
box.parm('divrate1').setExpression
      ('(((prim("../measureLot",0,"areaLot",0))/
      (detail("../avgarea","avgarea",0)))*3)+2')
box.parm('divrate2').setExpression
      ('(((prim("../measureLot",0,"areaLot",0))/
(detail("../avgarea","avgarea",0)))*2)+2')
...
```

Figura 136: Part del codi per a la creació de carrerons

Després esborrem tota la caixa menys la part d'abaix, de manera que ens quedem amb un pla amb un cert nombre de divisions. A continuació, deixem espais entre les primitives d'aquest pla (que seran els carrerons) fent-les més petites (Figura 137).

```
#delete all the points but the bottom ones
del4 = geo.createNode('delete')
del4.parm('entity').set(1) #change the entity type to points
del4.parm('groupop').set(2) #delete by expression
del4.parm('filter').setExpression("$TZ>$ZMIN") #all of the points
      #where the z component is greater than the minimum

#create disconnected primitives
poli = geo.createNode('polyextrude')
poli.setName('blockSize')
poli.setFirstInput(del4)
poli.parm('lsx').set(controlVariables.blockSizex)
poli.parm('lsy').set(controlVariables.blockSizey)
poli.parm('outputside').set(0)
...
```

Figura 137: Part del codi per a la cració de carrerons

A continuació creem un node TRANSFORM per a rotar-ho. Si en el fitxer XML hem dit que aquesta rotació ha de ser aleatòria, es farà servir la variable *stamp* (explicades a la secció 2.4.4)

corresponent a la primitiva actual. En el cas del patró grid, no es farà aquesta rotació, ja que volem que els carrerons siguin paral·lels a les avingudes principals (Figura 138).

```
#rotate the box
trans2 = geo.createNode('xform')
trans2.setName('rotation')
trans2.setFirstInput(poli2)
trans2.parm('px').setExpression("$CEX") #CE is the center
      # of the object
trans2.parm('py').setExpression("$CEY")
trans2.parm('pz').setExpression("$CEZ")
if controlVariables.pattern != 'grid':
      if controlVariables.streetsAngle == 'random':
            trans2.parm('rz').setExpression
                  ('360*(rand(stamp("../copy1","cy",0)))')
      else:
        trans2.parm('rz').setExpression
                  (controlVariables.streetsAngle)
...
```

Figura 138: Part del codi per a la creació de carrerons

Finalment, creem un node COOKIE que n'intersecarà el resultat amb la illa actual (referenciada per la variable *del3*), creant-hi els carrerons (Figura 139).

```
#intersect the current primitive with the array of boxes
cook = geo.createNode('cookie')
cook.setFirstInput(trans2)
cook.setInput(1,del3)
cook.parm('boolop').set(4) #change the operation to 'user defined'
cook.parm('outsideB').set(1)
cook.parm('tol3d').set(1.9404600607231259e-005) #to correct errors
```

Figura 139: Part del codi per a la creació de carrerons

6.4.2 Divisió parcel·les

Primer de tot dividirem la primitiva actual en un nombre de segments amb un node RESAMPLE. Aquest nombre serà proporcional al perímetre de la primitiva (de la mateixa

manera que ho hem fet amb els carrerons a la secció 6.4.1) i el calcularem dividint el perímetre de la primitiva actual per la mitjana del perímetre de totes les primitives, i multiplicant-ho pel nombre mitjà de segments que volem tenir. Cadascun d'aquests segments serà una parcel·la on més tard hi emplaçarem un edifici (Figura 140).

```
res3 = geo.createNode('resample')
res3.setFirstInput(mea3)
res3.parm('dolength').set(0)
res3.parm('dosegs').set(1)
res3.parm('segs').setExpression
      ('8 * prim("../measurePerim",0,"perim",0) /
      detail("../avgperim","avgperim",0)')
...
```

Figura 140: Part del codi per la creació de parcel·les

Amb el fi de crear el pati interior, amb un node POLIEXTRUDE crearem una primitiva igual que l'actual però més petita en una escala entrada al fitxer XML (Figura 141).

```
#the inner primitive
poli3 = geo.createNode('polyextrude')
poli3.setName('buildingSize')
poli3.parm('lsx').set(controlVariables.lotsWidth) #scale
poli3.parm('lsy').set(controlVariables.lotsWidth)
poli3.parm('outputside').set(0) #just output front
...
```

Figura 141: Part del codi per a la divició en parcel·les

Això ens crearia un pati interior regular, però volem que sigui de diferent mida per a cada edifici. Amb un node CARVE extraiem cada segment cap al pati interior segons la seva variable *stamp* (explicades a la secció 2.4.4) i el nombre total de punts de la primitiva, que podem trobar al node *point1* (Figura 142).

```
car = geo.createNode('carve')
car.setFirstInput(attp)
car.parm('secondu').set(1)
car.parm('domainu2').set(1)
car.parm('domainu1').setExpression
      ('3*stamp("../copy2","seg",0)/npoints("../point1")')
car.parm('domainu2').setExpression
      ('3*(stamp("../copy2","seg",0)+1)/npoints("../point1")')
...
```

Figura 142: Part del codi per a la creació de parcel·les

Finalment, unirem la cada segment de la primitiva exterior (referenciat per *carve*) amb el segment corresponent de la primitiva interior (refrenciat per pea2) amb un node MERGE i obligarem que tots els segments tornin a formar una sola primitiva fent servir un node SKIN (Figura 143).

```
#merge the inner and the outer primitives
mer2 = geo.createNode('merge')
mer2.setFirstInput(car)
mer2.setInput(1,pea2)

#join the segments
ski2 = geo.createNode('skin')
ski2.setFirstInput(mer2)
```

Figura 143: Part del codi per a crear parcel·les

6.4.3 Edificis

Primer caldrà crear un model d'edifici en un contenidor SOP apart per després insertar-ne copies en les diferents parcel·les que s'han creat.

6.4.3.1 Creació d'un edifici senzill

La base de l'edifici consistirà en un objecte de tipus BOX, les divisions *divrate1* i *divrate2* del qual representaran el nombre de plantes i de columnes respectivament (Figura 144).

```
theBox = geoContainer.createNode('box')
theBox.parm('type').set('polymesh')
theBox.parm('sizex').set(1)
theBox.parm('sizey').set(1)
theBox.parm('sizez').set(0.15)
theBox.parm('divrate1').set(4+1)
theBox.parm('divrate2').set(5+1)
theBox.parm('divrate3').set(2+1)
…
```

Figura 144: Part del codi per a la creació d'un edifici senzill

Per afegir-hi les textures, esborrem tota la geometria menys la part on volem col·locar una textura concreta. A continuació, amb un node UVQUICKSHADE indiquem el path del fitxer amb la imatge de la textura, la orientació i el nombre de vegades que es repetirà (Figura 145).

```
# ROOF ================================
dele1 = geoContainer.createNode('delete')
dele1.parm('negate').set('keep')
dele1.parm('groupop').set('filter')
dele1.parm('filter').setExpression("$TY==$YMAX")
dele1.setFirstInput(theBox)
uv1 = geoContainer.createNode('uvquickshade')
uv1.parm('texture').set(texturePath+'roof185.jpg')
uv1.parm('texture_axis').set(1)
uv1.parm('texture_su').setExpression("ch('../box1/divrate1')-1")#(4)
uv1.parm('texture_sv').setExpression("ch('../box1/divrate3')-1")#(2)
uv1.setFirstInput(dele1)
...
```

Figura 145: Part del codi per a la creació d'un edifici senzill

Finalment, unim totes les parts amb un node MERGE (Figura 146).

```
merge = geoContainer.createNode('object_merge')
merge.parm('numobj').set(4)
merge.parm("objpath1").set("../uvquickshade1")
merge.parm("objpath2").set("../uvquickshade2")
merge.parm("objpath3").set("../uvquickshade3")
merge.parm("objpath4").set("../uvquickshade4")
```

Figura 146: Part del codi per a la creació d'un edifici senzill

6.4.3.2 Insertar edificis a les parcel·les

Primer importarem el model d'edifici creat, guardat amb el nom de *building HDA* a la xarxa SOP principal (Figura 147).

```
theBuilding = geo.createNode('buildingHDA')
...
```

Figura 147: Part del codi per a la inserció d'edificis

Per a encaixar l'edifici en una parcel·la, utilitzem un node CREEP, explicat a la secció 2.4.5, passant-li com a primer paràmetre la geometria de l'edifici (referenciada per la variable *ed*) i com a segon paràmetre la parcel·la actual, referenciada per la variable *rev* (Figura 148).

```
creep = geo.createNode('creep')
creep.setFirstInput(ed)
creep.setInput(1,rev)
...
```

Figura 148: Part del codi per a la inserció d'edificis

Finalment, per a donar a cada edifici una alçada diferent, fem servir una expressió aleatòria utilitzant la variable stamp de la parcel·la actual (explicades a la secció 2.4.4) tant a un node TRANSFORM que augmentarà l'alçada total, com al node que conté la geometria de l'edifici per canviar-ne el nombre de plantes (Figura 149).

```
theBuilding.parm('boxHeight').setExpression
      ('2+floor(6*rand(stamp("../copy2","seg",0)))')
xform.parm('sz').setExpression
     ('(2+floor(6*rand(stamp("../copy2","seg",0))))*0.2')
```

Figura 149: Part del codi per la inserció d'edificis

6.5 Interfície d'usuari (Houdini Digital Asset)

Una vegada creada la geometria, podem implementar un Houdini Digital Asset (HDA), del que hem parlat a la secció 2.5, per a que actuï com a interfície per al usuari. Aquesta interfície permet controlar paràmetres "constructius" de la ciutat, però per a canviar, per exemple, els patrons, cal fer un altre XML i carregar-lo.

A la Figura 150 veiem com primer de tot cal crear una subxarxa amb la geometria que volem encapsular, on *container* és el node que la conté. A continuació hi creem el HDA i n'extreiem la definició.

```
citySubnet = container.collapseIntoSubnet
      (container.children(), subnet_name = 'citySubnet')
cityHDANode = citySubnet.createDigitalAsset(name = 'cityHDA',
      description = 'city HDA',
      ignore_external_references = True)
      cityHDANode.moveToGoodPosition()
cityDefinition = cityHDANode.type().definition()
```

Figura 150: Codi per a generar un HDA

El següent pas és crear la interfície de paràmetres. Per a cada paràmetre, creem una plantilla amb el nombre de components i valors per defecte i, a continuació, l'afegim a la definició del HDA, com es pot veure a la Figura 151.

```
citySize = hou.FloatParmTemplate(name='citySize',
            label='City scale', num_components=3,
            default_value=(1,1,1)
            )
      cityDefinition.addParmTuple(citySize,
            in_folder=("Params",), create_missing_folders=True)


[...]
```

Figura 151: Creació d'una plantilla de paràmetre per al HDA

Finalment, cal enllaçar aquests paràmetres amb els nodes corresponents de la subxarxa. Per fer-ho, necessitarem una funció auxiliar que ens busqui un node a partir del seu nom i el nom de la subxarxa on s'ha de trobar, que veiem a la Figura 152:

```
def findSubNode(searchSpace, nodeName):
      theNode = None
      for n in searchSpace.children():
            if (n.name() == nodeName):
                  theNode = n
      return theNode
```

Figura 152: Funció auxiliar per a trobar un node concret

Amb això, ja podem trobar el node i assignar-li els valors que l'usuari ens entri a través del HDA (Figura 153).

```
scale = findSubNode(cityHDANode,'scale')
      scale.parm("sx").setExpression('ch("../citySizex")')
      scale.parm("sy").setExpression('ch("../citySizey")')
      scale.parm("sz").setExpression('ch("../citySizez")')
...
```

Figura 153: Codi per a enllaçar un node amb el HDA

Una vegada fet això per tots els paràmetres que vulguem que siguin editables, el resultat és com el que veiem a la Figura 154.

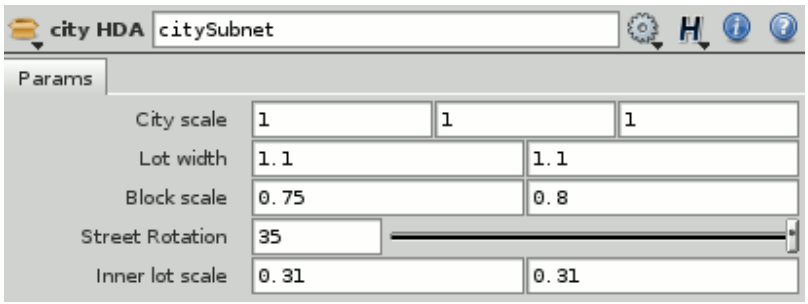


Figura 154: Aspecte final del HDA

Capítol 7: Resultats

Dins d'aquest capítol es mostren els resultats obtinguts del projecte. Els resultats estan visualitzats directament des de l'àrea de treball de Houdini, sense ser renderitzats per els motors del programa ja que el que s'ha buscat amb la realització del projecte ha sigut la generació de la geometria, no pas el resultat visual que dóna un motor de render.

7.1 Ciutat amb patró voronoi

Aquesta és el primer patró que es va implementar. Les bases teòriques han estat explicades a l'apartat 3.2.1. En aquest cas, utilitzem el següent mapa de densitat (Figura 155), que mostra una densitat més gran a la perifèria que al centre.

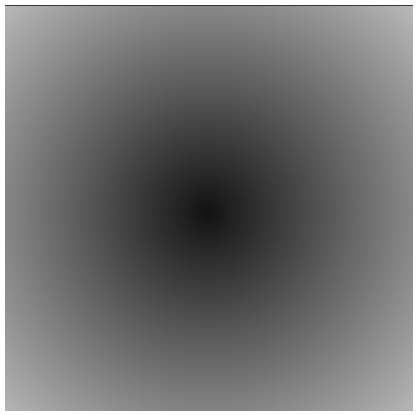


Figura 155: Mapa de densitat amb la població concentrada a la perifèria.

La xarxa VEX que produeix aquest mapa (Figura 156), efectivament genera un diagrama de Voronoi amb més cel·les que tendeixen a ser més concentrades a la perifèria. Es pot comparar amb el diagrama de Voronoi de la Figura 43, generat a partir del mapa de densitat oposat.

Figura 156: Diagrama de Voronoi generat a partir del mapa de la Figura 155

A partir d'aquest shader, després d'aplicar les operacions de compositing corresponents (explicades a l'apartat 4.2.4), obtenim la següent imatge:

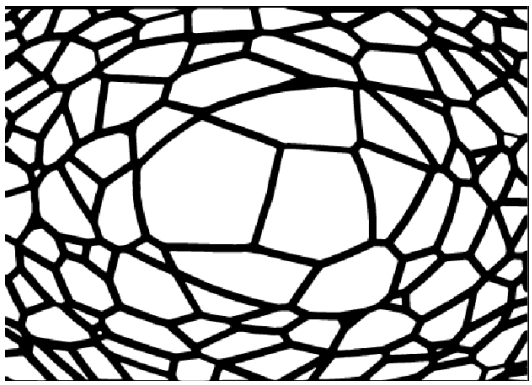


Figura 157: Imatge obtingutda després d'aplicar compositing a un voronoise

Una vegada obtinguda aquesta imatge, ja podem cridar les operacions que crearan la geometria. Primer de tot, però, cal crear el model d'edificis que poblaran la ciutat. Una vegada fet això, tindrem la xarxa SOP anomenada buildingSubnet (Figura 158). Aquesta xarxa serà igual en tots els exemples d'aquesta secció.

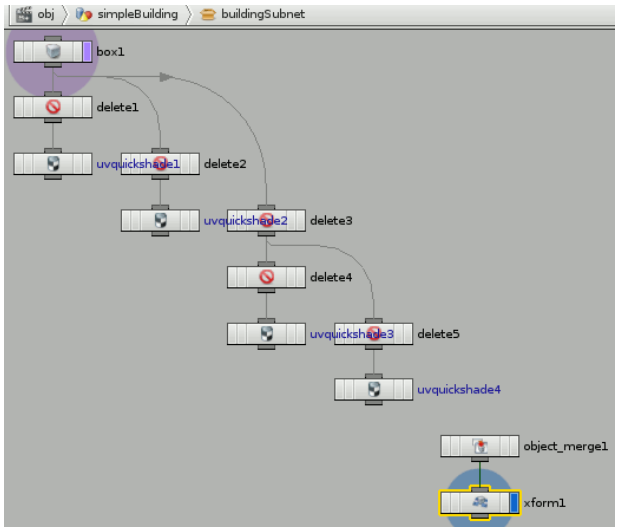


Figura 158: Xarxa buildingSubnet

A continuació es comencen a crear la geometria en una xarxa SOP a partir de la Figura 157, dividint-la en carrers i parcel·les on hi col·locarem els edificis generats per la subxarxa buildingSubnet. L'arbre de nodes resultant de totes aquestes operacions és el següent:

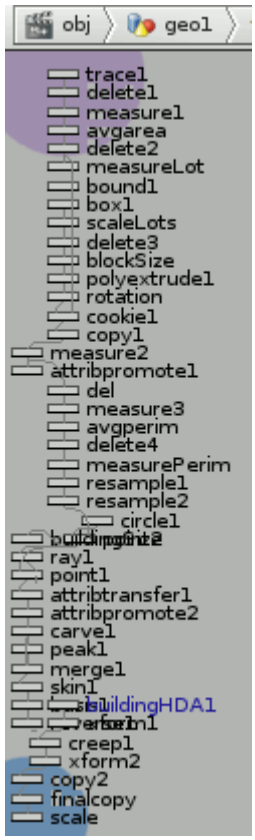


Figura 159: Arbre de nodes de la xarxa SOP

Aquesta xarxa de nodes genera el resultat que veiem en la Figura 160.

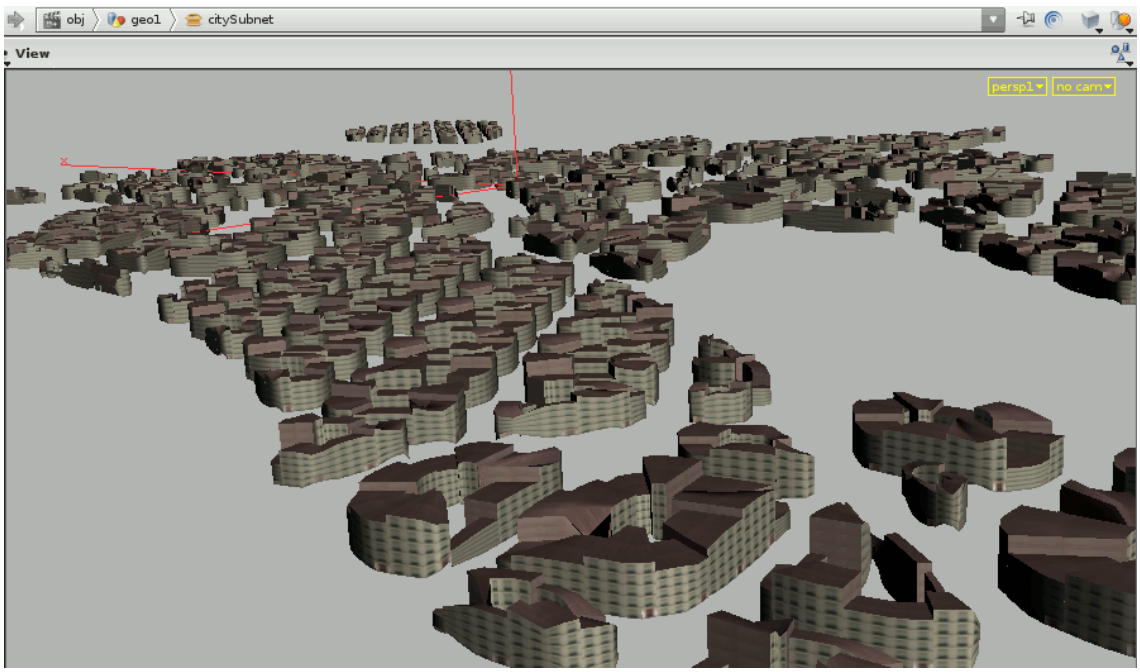


Figura 160: Resultat final de patró voronoise

7.2 Ciutat amb patró radial

Un exemple detallat del procés de creació d'una ciutat amb aquest patró es troba a la secció 5.3, amb el resultat final que es mostra a la Figura 161.

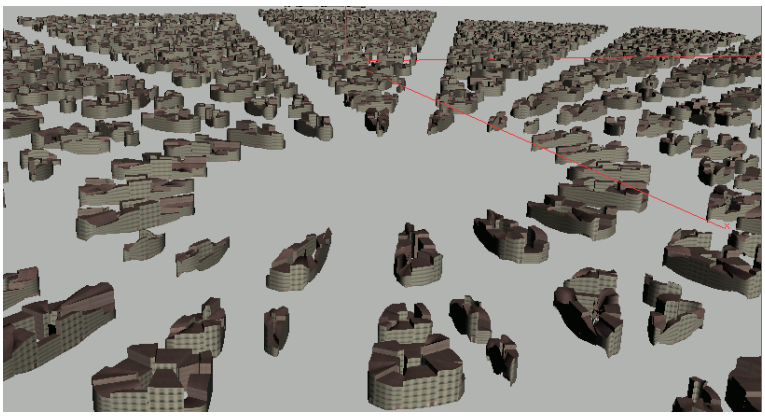


Figura 161: Resultat final de ciutat amb patró radial

7.3 Ciutat amb patró grid amb mapa d'ús del a terra

Els passos per generar una ciutat amb aquest patró seran molt semblants als del patró radial però, en aquest cas, també farem servir un mapa d'ús de la terra (Figura 162) per indicar en quines zones no és possible edificar.

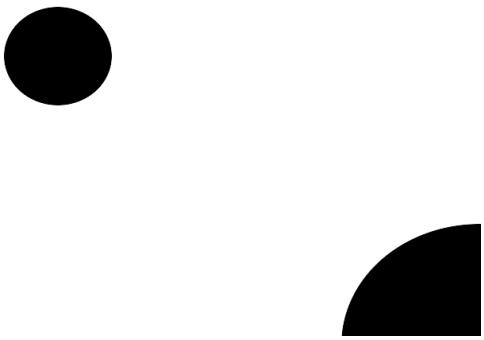


Figura 162: Mapa d'ús de la terra

En aquest cas, el compositing afegirà operacions per a esborrar les parts que al mapa estan indicades amb negre, obtenint el resultat de la Figura 163.

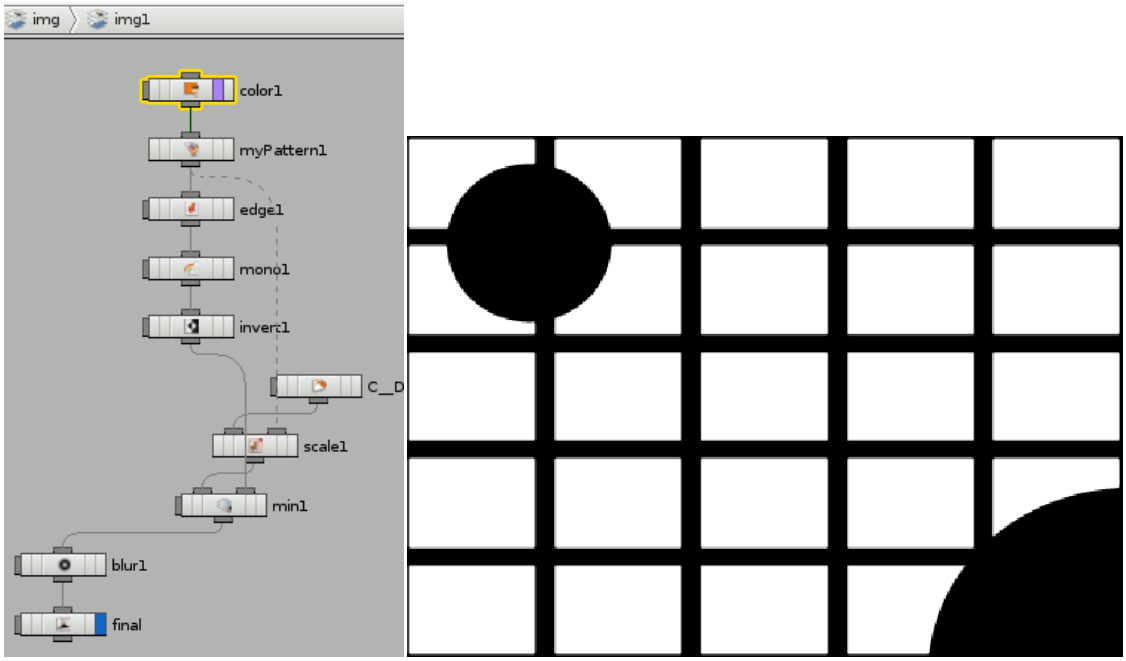


Figura 163: Xarxa compositing amb mapa d'ús de la terra

Finalment, es crea la geometria de la mateixa manera que s'ha fet amb els altres patrons, obtenint els resultats de la Figura 164.

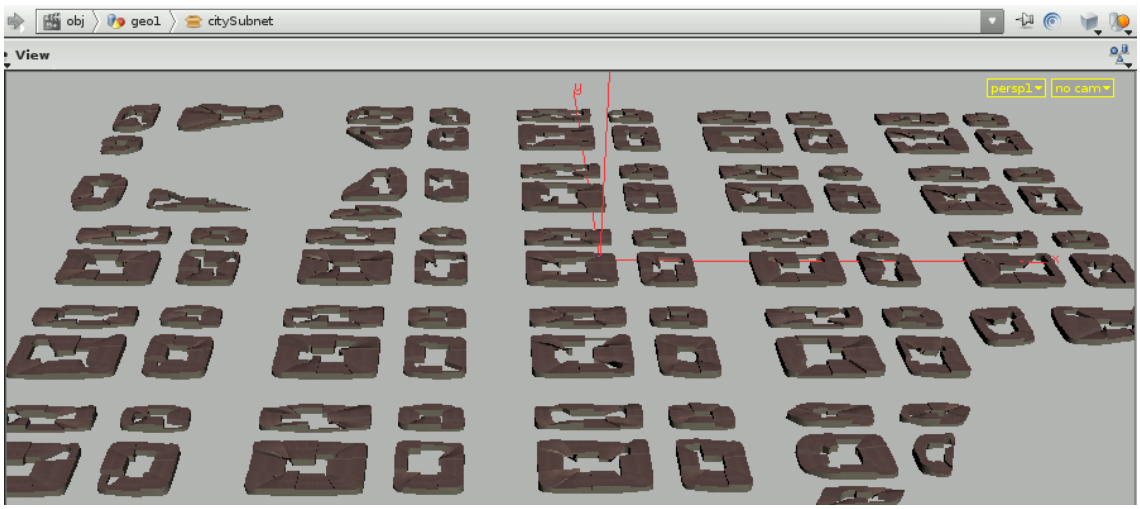


Figura 164: Resultat final de patró grid amb mapa d'ús de la terra

7.4 Ciutat amb districtes

Crearem una ciutat a partir del mapa de districtes de la Figura 165, on el color verd indica patró radial, el vermell indica voronoi i el blau indica grid.

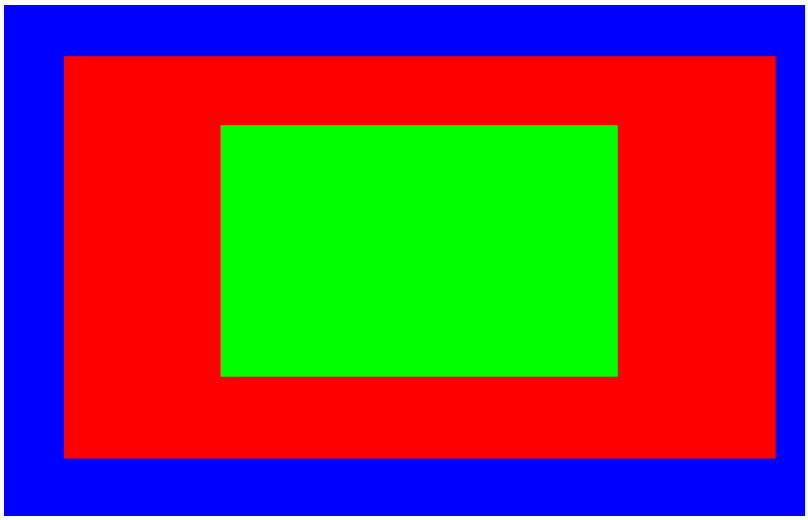


Figura 165: Mapa de districtes

Aquest mapa genera l'arbre de nodes VEX de la Figura 166. Hi podem observar com hi ha tres nodes IF (marcats en blau), que contenen les subxarxes amb els patrons corresponents.

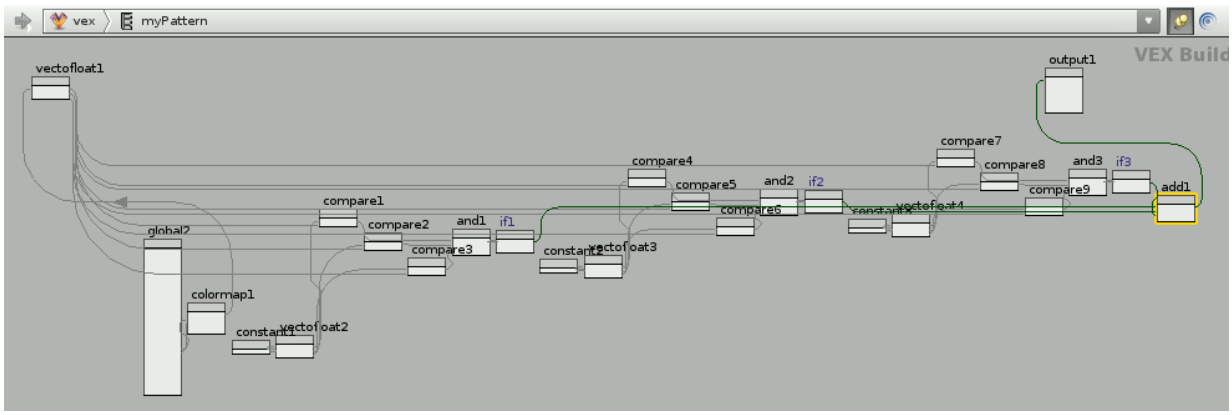


Figura 166: Xarxa VEX d'una ciutat amb districtes

Com a resultat d'aquest arbre, obtenim el shader de la Figura 167, on ja es pot observar com la part central és radial, la part mitjana és voronoi i la part perifèrica és grid.

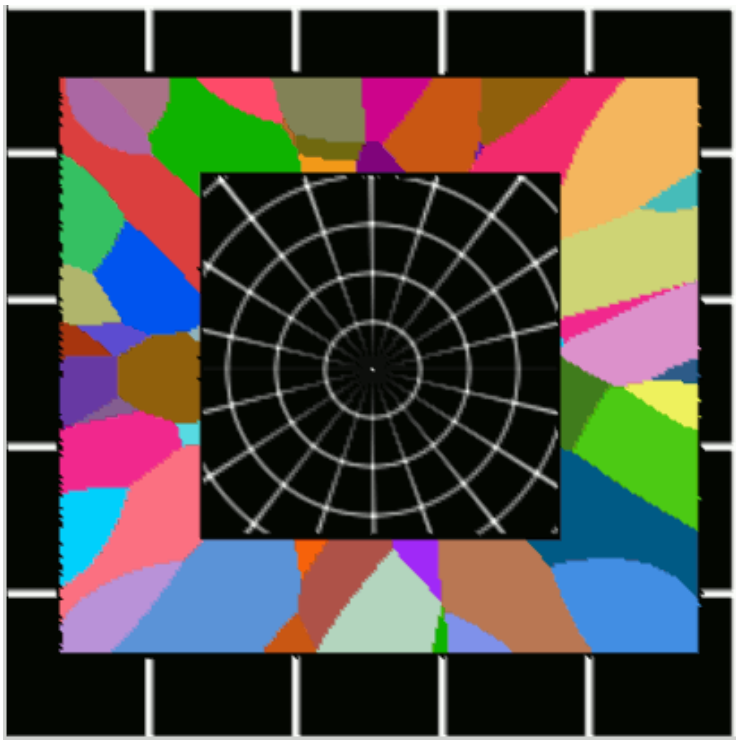


Figura 167: Shader obtingut a partir d'un mapa de districtes

En aquest cas, la operació de compositing, apart de ressaltar i binaritzar els marges del shader, també hi intercalarà una versió binaritzada del mapa de districtes per a afegir separació entre ells.

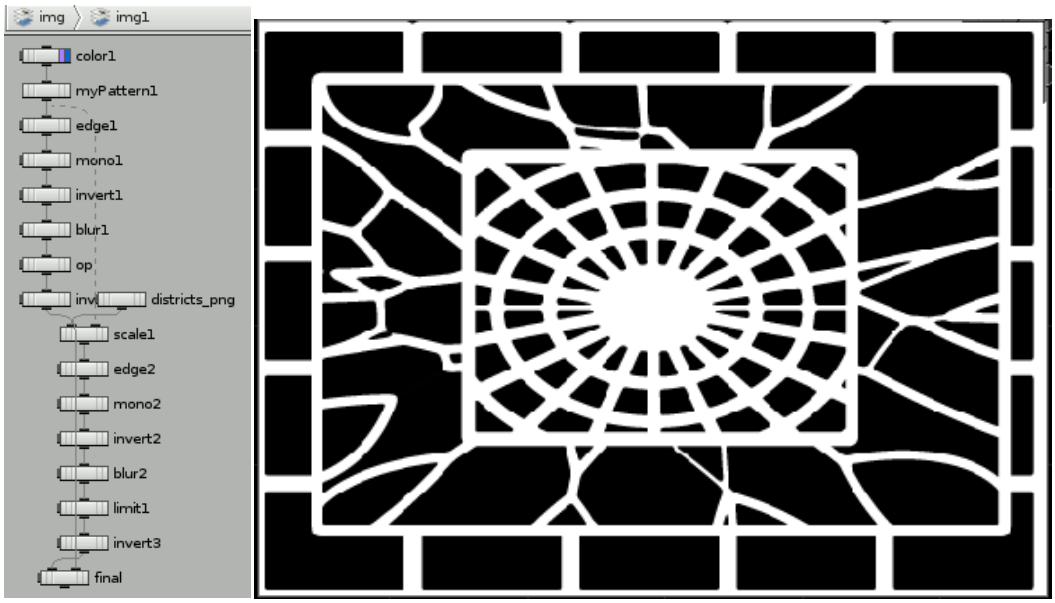


Figura 168: L'operació compositing generarà els nodes de l'esquerra, amb el resultat de la dreta

A partir d'aquesta imatge, la xarxa SOP resultant tindrà un arbre de nodes com el de la Figura 159, obtenint el resultat final de la Figura 169.

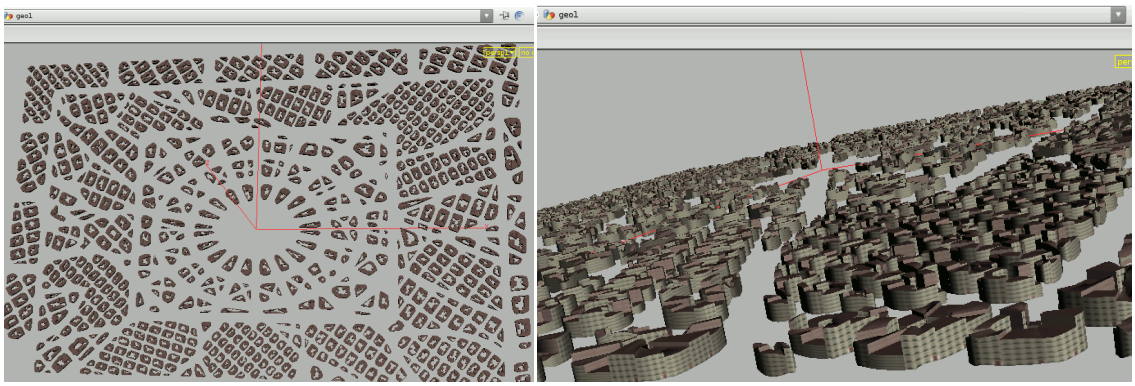


Figura 169: Resultat final de ciutat amb mapa de districtes

7.5 Ciutat a partir d'un mapa real

En aquest cas, ja hem vist que no calen xarxes VEX ni compositing. Farem servir el següent mapa, que correspon a Girona:

Figura 170: Mapa de Girona

Es crea directament una geometria a partir del mapa i es subdivideixen les illes en parcel·les on hi col·locarem els edificis. La subdivisió en carrerons no cal, ja que ens ve feta en el mapa. Obtenim el següent arbre de nodes en la xarxa SOP:

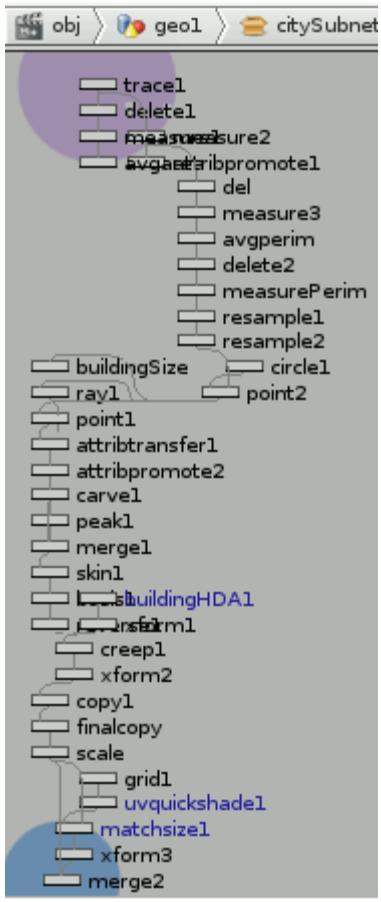


Figura 171: Xarxa de nodes SOP corresponent a generació de ciutats reals

Aquesta xarxa de nodes genera el següent resultat final:

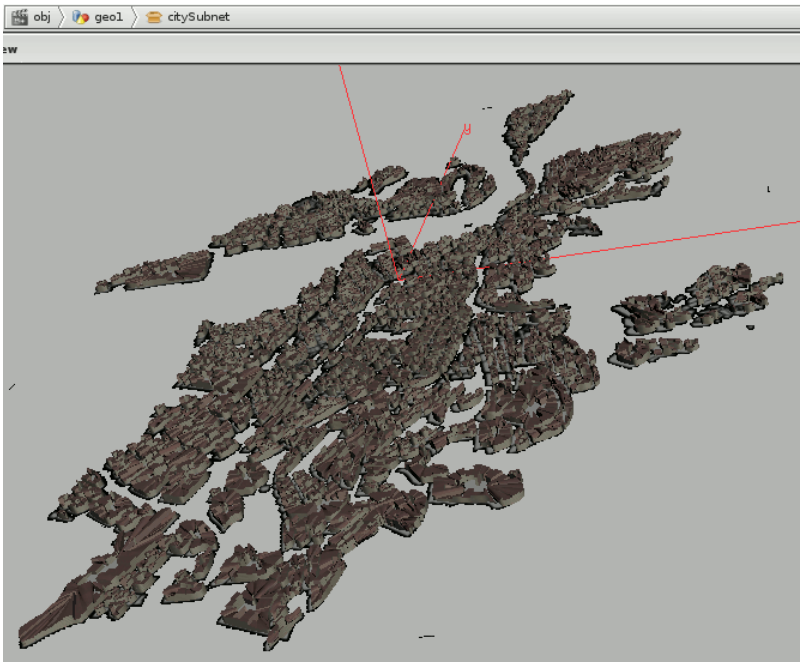


Figura 172: Resultat final de ciutat amb mapa de Girona

7.6 Houdini Digital Asset

En aquesta secció parlarem de l'ús del HDA, del que hem explicat la seva implementació a la secció 6.5.

7.6.1 Paràmetre City scale

El primer paràmetre que l'usuari podrà modificar és *City scale*, que controla la mida total de la ciutat. Aquest paràmetre sempre valdrà per defecte (1,1,1). Prenem com a base la ciutat de la Figura 173.

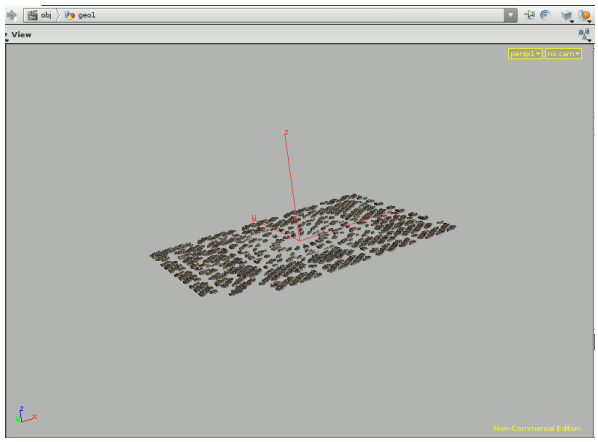


Figura 173: Ciutat amb escala (1,1,1)

Primer canviem el paràmetre a (1,2,1), de manera que la ciutat resultant serà el doble de gran per l'eix de les y, com es pot observar a la Figura 174.

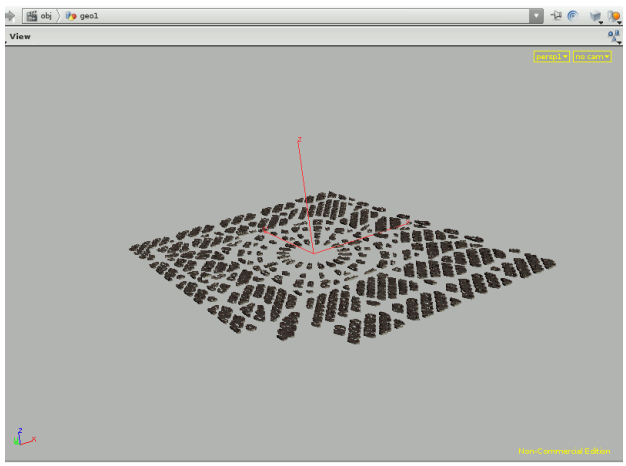


Figura 174: Ciutat amb escala (1,2,1)

Podem fer que la ciutat també creixi per l'eix de les x canviant el paràmetre a (2,2,1), obtenint el resutat de la Figura 175.

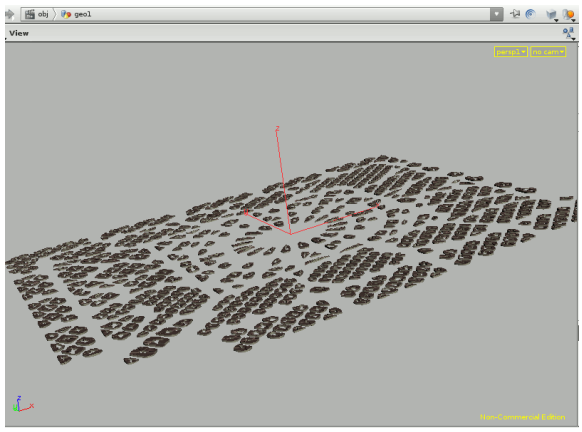


Figura 175: Ciutat amb escala (2,2,1)

Finalment, podem controlar l'alçada general dels edificis. Canviem el paràmetre *City scale* a (2,2,5), de manera que tots els edificis seran 5 vegades més grans, com es pot veure a la Figura 176.

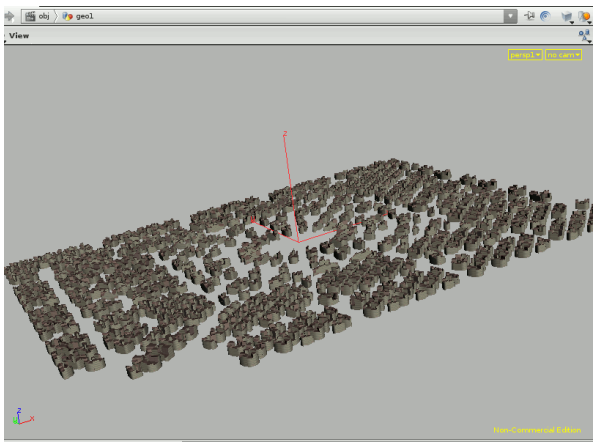


Figura 176: Ciutat amb escala (2,2,5)

7.6.2 Paràmetre Lot width

Aquest paràmetre permet modificar l'amplada de les illes, que sempre valdrà per defecte (1.1, 1.1) i només estarà disponible en el cas de generació de ciutats sintètiques. Com a base per aquest exemple, tenim les illes de la Figura 177.

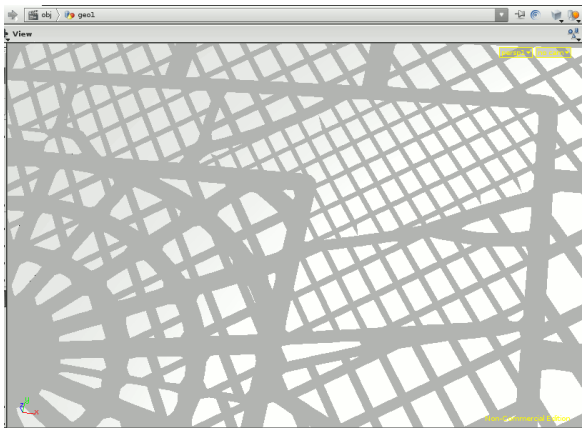


Figura 177: Illes amb escalat (1.1,1.1)

Podem fer que l'amplada general de les illes sigui més gran canviant aquest paràmetre a (3,3,), obtenint el resultat de la Figura 178.

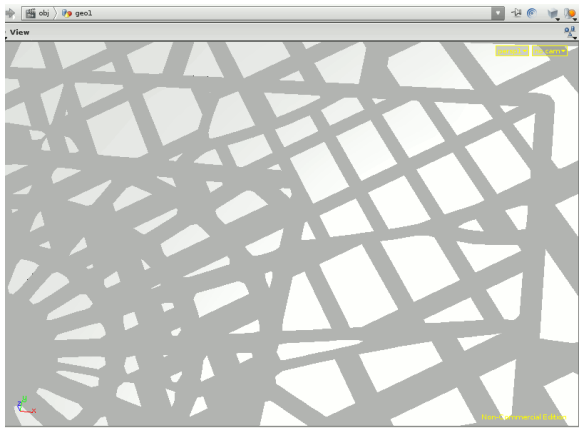


Figura 178: Illes amb escalat (3,3)

7.6.3 Paràmetre Block scale

Aquest paràmetre controla la grandària dels blocs en detriment dels carrerons. Primerament l'haurem introduit al XML com als paràmetres *blocksizex* i *blocksizey* de l'acció *divideStreets*. Només serà accessible en el cas de generació de ciutats sintètiques.

En l'exemple de la Figura 179, *Block scale* val (0.75, 0.8).

Figura 179: Block scale (0.75,0.8)

Volem que els carrerons siguin més amples, així que canviem el paràmetre a (0.6,0.6), obtenint el resultat de la Figura 180.

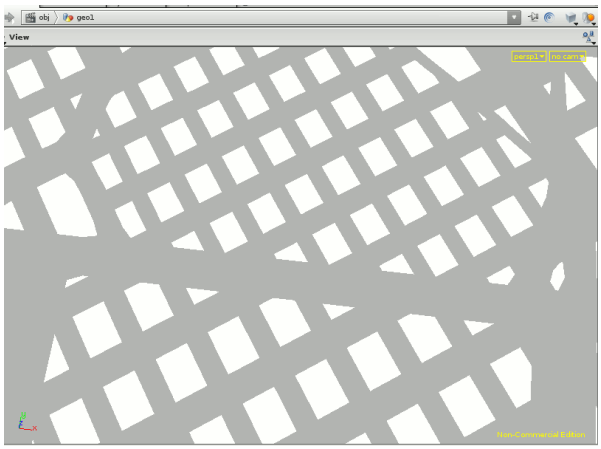


Figura 180: Block scale (0.6,0.6)

Si el paràmetre val (1,1), els carrerons desapareixen, com es pot veure a la Figura 181.

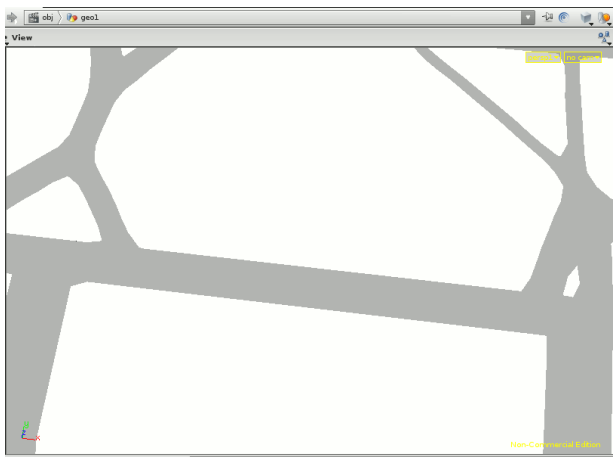


Figura 181: Block scale (1,1)

7.6.4 Paràmetre Street Rotation

Aquest paràmetre controla la orientació dels carrerons. Primerament l'haurem entrat en el XML com a paràmetre *angle* de la acció *divideStreets*, i només serà accesible en el cas que haguem entrat un valor numèric en aquest paràmetre. Si hi haviem entrat *random*, no ho podrem canviar després.

A la Figura 182 hi veiem un conjunt d'illes amb una orientació inicial de 35º.

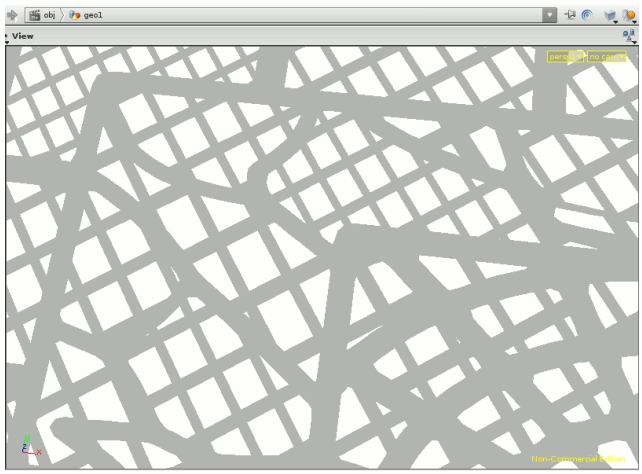


Figura 182: Illes amb orientació de 35º

Si canviem el paràmetre Street Rotation i li diem que la orientació ha de ser de 90º, obtenim el resutat de la Figura 183.

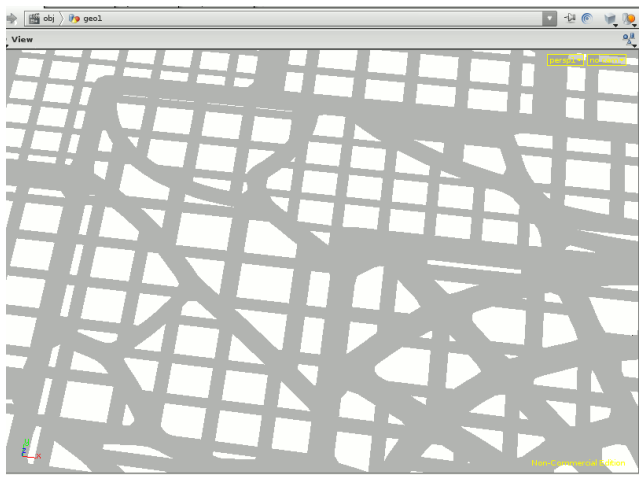


Figura 183: Illes amb orientació de 90º

7.6.5 Paràmetre Inner lot scale

Aquest paràmetre controla la mida del pati interior. Primerament l'haurem introduit al fitxer XML com a paràmetre *width* de l'acció *makeLots*.

A la Figura 184 hi veiem un bloc d'edificis, el valor del paràmetre *Inner lot scale* dels quals és (0.31,0.31).

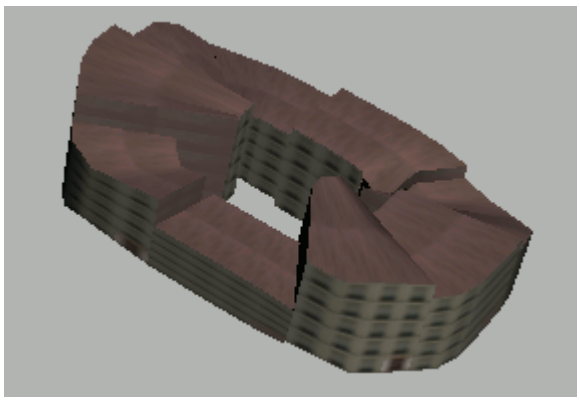


Figura 184: Bloc amb pati interior d'escala (0.31,0.31)

Si volem que el pati interior sigui més gran i, per tant, els edificis més estrets, augmentem el paràmetre. En el cas de la Figura 185, l'hem posat a (0.5,0.5).

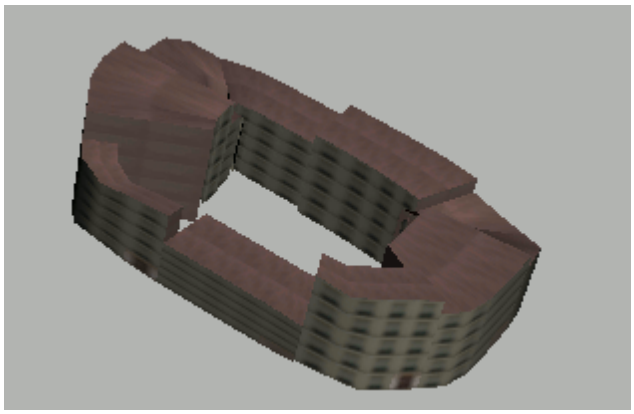


Figura 185: Bloc amb pati interior de mida (0.5,0.5)

7.7 Temps d'execució

Cal diferenciar entre el temps que triga el codi a executar-se, que és el que es triga a crear totes les xarxes de nodes necessàries, i el temps que triga el Houdini a dibuixar el resultat final quan volem visualitzar-lo per primera vegada.

Com veiem a la taula de la Figura 186, el temps d'execució del codi és molt petit i gairebé igual en tots els casos. En el cas de la generació de la ciutat de Girona, que és l'exemple amb més quantitat de geometria que s'ha provat, aquest temps és inclús més petit, cosa que fa deduïr que no depén de la geometria que generarà, sinó de la quantitat i complexitat de les xarxes de nodes creades, que en el cas de les ciutats reals és menor.

Per altra banda, es pot observar que el temps que triga el Houdini a dibuixar les diferents ciutats sí que és proporcional a la quantitat de geometria generada, sent l'exemple de patró grid amb mapa d'ús de la terra el més senzill (11 segons) i l'exemple de la ciutat de Girona el més complexe (34 segons).

També es pot observar per la mida dels fitxers OBJ resultants que es genera una quantitat molt gran de geometria en molt pocs segons, que demanaria hores si hagués de ser generada a mà.

	Execució codi	Creació geometria	Mida fitxer OBJ (MB)
Patró voronoise (Figura 160)	4 segons	17 segons	22.4
Patró radial (Figura 161)	4 segons	26 segons	43.9
Patró grid amb ús de la terra (Figura 164)	4 segons	11 segons	6.7
Districtes (Figura 169)	4 segons	33 segons	53.6
Girona (Figura 172)	3 segons	34 segons	64.5

Figura 186: Temps d'execució per a diferents ciutats

Capítol 8: Conclusions

Els objectius (secció 1.1) i abast (secció 1.2) d'aquest projecte s'han pogut assolir satisfactòriament. Aquests objectius eren els següents:

- Aplicació d'un conjunt de regles que generi iterativament la xarxa de carrers.

  S'ha definit un sistema de regles (XMLShape) per a generar xarxes de carrers que, a més, divideixen les illes resultants en parcel·les i permeten aixecar-hi edificis.

  Aquest sistema ens permet definir, a més de triar quin tipus de ciutat volem generar, tota una sèrie de paràmetres per a totes les fases de la generació. Així, podem triar paràmetres generals de cada patró, com ara on estarà ubicat el centre de la ciutat i la freqüència de les illes, i paràmetres més específics de cada geometria, com ara l'amplada dels carrerons o la mida dels patis interiors dels edificis.

  Tot això ho aconseguim amb l'avantatge que suposa que el XML sigui un llenguatge senzill i conegut per l'usuari.

- Aplicacions d'aquest sistema per generar ciutats a partir del mapa d'una ciutat real.

  Podem generar geometria a partir d'un mapa d'una ciutat real, a més de poder-hi posar una textura al terra. En fer servir una imatge que ens dóna l'usuari, el sistema resulta ser molt genèric i configurable, ja que qualsevol imatge amb dos colors és vàlida per a ser usada per aquest propòsit.

  El sistema genera una geometria a partir d'aquest mapa, la divideix en carrerons i parcel·les i hi aixeca edificis, resultant en una ciutat artificial amb la mateixa estructura d'una ciutat que existeix a la realitat.

- Aplicacions per generar ciutats sintètiques amb mapes de districtes, ús de la terra i densitat de població com a entrades.

  Podem generar ciutats sintètiques a través d'aquestes entrades. Amb un mapa de districtes podem definir fronteres a la ciutat i fer que aquesta consti de diferents patrons en cada part. Amb un mapa d'ús de la terra podem definir en quines parts de la ciutat no s'hi pot edificar a causa de rius, llacs, muntanyes, etc. Amb un mapa de densitat podem definir les diferents densitats de població a través de la ciutat.

  A més, hem implementat tres patrons diferents per a les ciutats sintètiques:

- Voronoise: Crea una xarxa de carrers principals a partir d'un diagrama de Voronoi i una funció de densitat, que pot ser centrada o a partir d'un mapa.
- Radial: Crea una xarxa de carrers principals radial, amb un punt central que podem col·locar a qualsevol lloc de la ciutat.
- Grid: Crea una xarxa de carrers principals en forma de graella, amb freqüència de carrers verticals i horitzontals variable.

Una vegada obtinguda la estructura de la xarxa de carrers principal, creem la geometria bàsica. Després hi creem els carrerons, formant blocs més petits. Finalment dividim aquests blocs en parcel·les per aixecar-hi els edificis.

Apart de tot això, també s'ha desenvolupat un HDA (Houdini Digital Asset), que actua com a interfície d'usuari. Aquest, un cop generada la ciutat, ens permet modificar-ne alguns paràmetres que canviaran immediatament la geometria. Aquesta eina permet que alguns dels paràmetres entrats per l'usuari en el fitxer XML no hagin de ser definitius i que aquest pugui fer canvis sense haver d'escriure un altre fitxer XML i tornar a executar el programa.

Capítol 9: Treball futur

A partir del treball fet en aquest projecte es podrien implementar un gran nombre de millores i ampliacions.

Una millora que s'hi podria fer és la manera com es genera una ciutat amb districtes. Tal i com està fet, s'ha de crear una xarxa VEX força complexa amb subxarxes per a cada patró i la necessitat de comparar les tres components RGB per a cadascun d'ells. Es podria obtenir el mateix resultat de manera més eficient, per exemple creant una xarxa VEX separada per a cada patró i fer-ne la composició segons districtes a nivell COP2.

Pel que fa a les ampliacions, les possibilitats són molt grans. Per exemple, a partir del projecte OpenStreetMap (www.openstreetmap.org) es podrien generar mapes per a poder-los tractar posteriorment amb el nostre codi.

També seria interessant poder generar proceduralment elements que poblin els carrers, com ara cotxes, arbres, etc.

Finalment, es podria afegir l'element temps a les ciutats, de manera que podriem veure el seu procés de creixement al llarg dels anys.

Capítol 10: Bibliografia

2. Lindenmayer, P. i. (1991). Sistemes de Lindenmayer.
3. Müller, P.; Parish, Y. I. (2001). Procedural Modeling of Cities. *Siggraph* .
4. Sun, J.; Baciu, G.; Yu, X.; Green, M. (2002). Template-Based Generation of Raoad Networks for Virtual City Modeling. *VRST* .
5. Müller, P; Wonka, P (2006). Procedural Modeling of Buildings. *Siggraph*
6. Documentació Houdini: http://www.sidefx.com/docs/houdini9.1/hom/
7. Tutorial Python: http://docs.python.org/3.0/tutorial/
8. Tutorial XML DOM: http://www.w3schools.com/dom/default.asp
9. World Wide Web Consortium: http://www.w3.org/

Capítol 11: Annexos

11.1 Codi XML per al patró voronoise amb densitat centrada

```
<XMLShape>

	<header type="XMLShape:streetNetwork" />

	<rule id="1">
		<predecessor name="none" />
		<successor>
			<action name="voronoise" params="4">
				<param name="frequency" value="15" />
				<param name="cx" value="0.5" />
				<param name="cy" value="0.5" />
				<param name="density" value="centered" />

				<product name="ground" />
			</action>
		</successor>
	</rule>

	<rule id="2">
		<predecessor name="ground" />
		<successor>
			<action name="landUse" params="1">
				<param name="waterMap" value="None" />

				<product name="streets" />
			</action>
		</successor>
	</rule>

	<rule id="3">
		<predecessor name="streets" />
		<successor>
			<action name="divideStreets" params="3">
				<param name="angle" value="random" />
				<param name="blockSizex" value="0.75" />
				<param name="blockSizey" value="0.8" />

				<product name="block" />
			</action>
		</successor>
	</rule>

	<rule id="4">
		<predecessor name="block" />
		<successor>
			<action name="makeLots" params="1">
				<param name="width" value="0.31" />

				<product name="lot" />
			</action>
		</successor>
	</rule>

	<rule id="5">
		<predecessor name="lot" />
		<successor>
			<action name="createBuildings" params="1">
				<param name="texturePath" value="path textures" />

				<product name="building" />
			</action>
		</successor>
	</rule>

</XMLShape>
```

11.2 Codi XML per al patró voronoise amb mapa de densitats

```
<XMLShape>

	<header type="XMLShape:streetNetwork" />

	<rule id="1">
		<predecessor name="none" />
		<successor>
			<action name="voronoise" params="4">
				<param name="frequency" value="15" />
				<param name="cx" value="0.5" />
				<param name="cy" value="0.5" />
				<param name="density" value="path mapa densitat" />

				<product name="ground" />
			</action>
		</successor>
	</rule>

	<rule id="2">
		<predecessor name="ground" />
		<successor>
			<action name="landUse" params="1">
				<param name="waterMap" value="None" />

				<product name="streets" />
			</action>
		</successor>
	</rule>

	<rule id="3">
		<predecessor name="streets" />
		<successor>
			<action name="divideStreets" params="3">
				<param name="angle" value="random" />
				<param name="blockSizex" value="0.75" />
				<param name="blockSizey" value="0.8" />

				<product name="block" />
			</action>
		</successor>
	</rule>

	<rule id="4">
		<predecessor name="block" />
		<successor>
			<action name="makeLots" params="1">
				<param name="width" value="0.31" />

				<product name="lot" />
			</action>
		</successor>
	</rule>

	<rule id="5">
		<predecessor name="lot" />
		<successor>
			<action name="createBuildings" params="1">
				<param name="texturePath" value="path textures" />

				<product name="building" />
			</action>
		</successor>
	</rule>

</XMLShape>
```

11.3 Codi XML per al patró radial

```
<XMLShape>

	<header type="XMLShape:streetNetwork" />

	<rule id="1">
		<predecessor name="none" />
		<successor>
			<action name="radial" params="4">
				<param name="freqRadial" value="15" />
				<param name="freqAngular" value="3" />
				<param name="cx" value="0.95" />
				<param name="cy" value="0.95" />

				<product name="ground" />
			</action>
		</successor>
	</rule>

	<rule id="2">
		<predecessor name="ground" />
		<successor>
			<action name="landUse" params="1">
				<param name="waterMap" value="None" />

				<product name="streets" />
			</action>
		</successor>
	</rule>

	<rule id="3">
		<predecessor name="streets" />
		<successor>
			<action name="divideStreets" params="3">
				<param name="angle" value="random" />
				<param name="blockSizex" value="0.75" />
				<param name="blockSizey" value="0.8" />

				<product name="block" />
			</action>
		</successor>
	</rule>

	<rule id="4">
		<predecessor name="block" />
		<successor>
			<action name="makeLots" params="1">
				<param name="width" value="0.31" />

				<product name="lot" />
			 </action>
		</successor>
	</rule>

	<rule id="5">
		<predecessor name="lot" />
		<successor>
			<action name="createBuildings" params="1">
				<param name="texturePath" value="path textures" />

				<product name="building" />
			</action>
		</successor>
	</rule>

</XMLShape>
```

11.4 Codi XML per al patró grid amb mapa d'ús de la terra

```
<XMLShape>

	<header type="XMLShape:streetNetwork" />

	<rule id="1">
		<predecessor name="none" />
		<successor>
			<action name="grid" params="4">
				<param name="freqx" value="5" />
				<param name="freqy" value="5" />
				<param name="cx" value="0.5" />
				<param name="cy" value="0.5" />

				<product name="ground" />
			</action>
		</successor>
	</rule>

	<rule id="2">
		<predecessor name="ground" />
		<successor>
			<action name="landUse" params="1">
				<param name="waterMap" value="path mapa ús terra" />

				<product name="streets" />
			</action>
		</successor>
	</rule>

	<rule id="3">
		<predecessor name="streets" />
		<successor>
			<action name="divideStreets" params="3">
				<param name="angle" value="45" />
				<param name="blockSizex" value="0.75" />
				<param name="blockSizey" value="0.8" />

				<product name="block" />
			 </action>
		</successor>
	</rule>

	<rule id="4">
		<predecessor name="block" />
		<successor>
			<action name="makeLots" params="1">
				<param name="width" value="0.31" />

				<product name="lot" />
		    </action>
		</successor>
	</rule>

	<rule id="5">
		<predecessor name="lot" />
		<successor>
			<action name="createBuildings" params="1">
				<param name="texturePath" value="path textures"/>

				<product name="building" />
			</action>
		</successor>
	</rule>
```

```
</XMLShape>
```

11.5 Codi XML per a ciutat amb districtes

```
<XMLShape>

        <header type="XMLShape:streetNetwork" />

        <rule id="1">
                <predecessor name="none" />
                <successor>
                        <action name="districts" params="5">
                                <param name="districtMap" value="path mapa districtes" />
                                <param name="angle" value="random" />

                                <param name="colorDef1" value="red" />
                                <product name="voronoiseDistrict" />

                                <param name="colorDef2" value="green" />
                                <product name="gridDistrict" />

                                <param name="colorDef3" value="blue" />
                                <product name="radialDistrict" />
                    </action>
                </successor>
        </rule>

        <rule id="2">
                <predecessor name="voronoiseDistrict" />
                <successor>
                        <action name="voronoise" params="4">
                                <param name="density" value="centered" />
                                <param name="frequency" value="15" />
                                <param name="cx" value="0.5" />
                                <param name="cy" value="0.5" />

                                <product name="ground" />
                        </action>
                </successor>
        </rule>

        <rule id="3">
                <predecessor name="gridDistrict" />
                <successor>
                        <action name="grid" params="4">
                                <param name="freqx" value="5" />
                                <param name="freqy" value="5" />
                                <param name="cx" value="0.5" />
                                <param name="cy" value="0.5" />

                                <product name="ground" />
                        </action>
                </successor>
        </rule>

        <rule id="4">
                <predecessor name="radialDistrict" />
                <successor>
                        <action name="radial" params="4">
                                <param name="freqRadial" value="15" />
                                <param name="freqAngular" value="3" />
                                <param name="cx" value="0.75" />
                                <param name="cy" value="0.75" />

                                <product name="ground" />
                        </action>
```

```
		</successor>
	</rule>

	<rule id="5">
	<predecessor name="ground" />
		<successor>
			<action name="landUse" params="1">
				<param name="waterMap" value="None" />

				<product name="streets" />
			</action>
		</successor>
	</rule>

	<rule id="6">
		<predecessor name="streets" />
		<successor>
		<action name="divideStreets" params="3">
				<param name="angle" value="random" />
				<param name="blockSizex" value="0.75" />
				<param name="blockSizey" value="0.8" />

				<product name="block" />
		    </action>
		</successor>
	</rule>

	<rule id="7">
		<predecessor name="block" />
		<successor>
			<action name="makeLots" params="1">
				<param name="width" value="0.31" />

				<product name="lot" />
			</action>
		</successor>
	</rule>

	<rule id="8">
		<predecessor name="lot" />
		<successor>
			<action name="createBuildings" params="1">
				<param name="texturePath" value="path textures" />

				<product name="building" />
			</action>
		</successor>
	</rule>

</XMLShape>
```

11.6 Codi XML per a generació de ciutat real

```
<XMLShape>

	<header type="XMLShape:streetNetwork" />

	<rule id="1">
		<predecessor name="none" />
		<successor>
			<action name="loadMap" params="2">
				<param name="file" value="path mapa ciutat" />
				<param name="floor" value="path terra ciutat" />

				<product name="block" />
```

```
                </action>
            </successor>
        </rule>

        <rule id="2">
            <predecessor name="block" />
            <successor>
                <action name="makeLots" params="1">
                    <param name="width" value="0.31" />

                    <product name="lot" />
                </action>
            </successor>
        </rule>

        <rule id="3">
            <predecessor name="lot" />
            <successor>
                <action name="createBuildings" params="1">
                    <param name="texturePath" value="path textures" />

                    <product name="building" />
                </action>
            </successor>
        </rule>
</XMLShape>
```