Creació procedural d'edificis

Treball final de grau

Autor: Roger Hernando Buch
Especialitat: Computació
Titulació: Grau en enginyeria informàtica

Director: Antonio Chica Calaf
Departament: Llenguatges i Sistemes Informàtics

Centre: Facultat d'informàtica de Barcelona(FIB)
Universitat: Universitat Politecnica de Catalunya(UPC)
Data de defensa: 20 de Juny del 2013

Índex

Abstract **3**
0.1 Català 3
0.2 Español 3
0.3 English 4

1 Introducció **5**
1.1 Formulació del problema 5
1.2 Enfoc del problema 6
1.2.1 Gramàtica 6
1.2.2 Parser 6
1.2.3 Generador 7
1.2.4 Visualitzador/Exportador 7

2 Competències tècniques **8**
2.1 CCO1.1 8
2.2 CCO1.2 9
2.3 CCO2.6 9

3 Conceptes bàsics **10**
3.1 Generació procedural 10
3.2 Gramàtiques 11
3.2.1 Sistemes-L 12
3.2.2 Gramàtiques de forma 12
3.2.3 CGA Shape grammar 13
3.3 Representació del models 14
3.4 Format ply 15

4 Solució proposada **17**
4.1 El llenguatge 18
4.1.1 Tipus de dades 19
4.1.2 Prioritat 19

4.1.3 Regles 20
4.1.3.1 Expressions 21
4.1.3.2 Regles de substitució 21
4.1.3.3 Regles de separació 23
4.1.3.4 Regles de repetició 25
4.1.3.5 Regles de separació de components 26
4.1.4 Exemple de generació 27

5 Implementació **28**
5.1 Parser més Generador 28
5.1.1 Parser 29
5.1.2 Generador 30
5.1.2.1 Generació de paraules 30
5.1.2.2 Format 32
5.1.3 Organització del sistema 33
5.2 Visualitzador 35
5.2.1 Funcionalitats 35
5.2.2 Organització del visualitzador 36
5.2.3 Exemples 37

6 Planificació **39**
6.1 Planificació inicial 39
6.2 Planificació real 41
6.3 Metodologia 41

7 Anàlisi de Costos **43**
7.1 Costos de hardware 43
7.2 Costos software 44
7.3 Costos de personal 44
7.4 Costos de despeses generals 45
7.5 Costos totals 46
7.6 Anàlisi del cost mediambiental 46

8 Conclusions **47**
8.1 Treball futur 48

9 Bibliografia **49**

Abstract

0.1 Català

L'objectiu d'aquest projecte, és explorar un mètode de generació procedural basat en regles, amb l'objectiu de generar models tridimensionals d'edificis, automàticament a partir d'un conjunt de regles definides per l'usuari.

S'ha escollit la gramàtica CGA shape, introduïda per *Peter Wonka* i *Pascal Müller*[1]. Amb els exemples i les proves dutes a terme, la gramàtica ha demostrat ser robusta i capaç de generar multitud de models diferents, valent-se d'un ventall de regles prou variades, i de la possibilitat d'importar models complexos per afegir detall a l'objecte generat.

Per aconseguir generar edificis a partir d'aquest gramàtica, s'han dissenyat dos sistemes software, un que implementa les tasques de parsing i generació, i l'altre que implementa les tasques de vehiculització, i exportació dels models en un format estàndard, per poder ser usats en altres aplicacions.

0.2 Español

El objetivo de este proyecto, es explorar un método de generación procedural basado en reglas, con el objetivo de generar modelos tridimensionales de edificios automáticamente a partir de un conjunto de reglas definidas por el usuario.

Se ha escogido la gramática CGA shape, introducida por *Peter Wonka* y *Pascal Müller*[1]. Con los ejemplos y las pruebas realizadas, la gramática ha demostrado ser robusta y capaz de generar multitud de modelos distintos, valiéndose de un abanico de reglas suficientemente amplio, y de la posibilidad de importar modelos complejos para añadir detalle al objecto generado.

Para conseguir generar edificios a partir de esta gramática, se han diseñado dos sistemas software, uno que implementa las tareas de parsing y generación, y el otro implementa las tareas de visualización, y exportación de los modelos en un formato estándar, para ser usados en otras aplicaciones.

0.3 English

The aim of this project, is to explore a procedural generation method based on rules, in order to generate three-dimensional models of buildings, automatically just reading a set of rules defined by the user.

CGA shape grammar has been chosen to be used in this project, this grammar was first-time introduced by *Peter Wonka* and *Pascal Müller*[1]. With the examples and the tests done. The CGA grammar has been proved as a robust grammar which is capable of generating many different models, using sufficient range of rules, and the ability to import complex meshes to add detail to the generated models.

To archive the objective of the project which is be able to generate buildings using the grammar, two software systems have been designed, one that implements the parsing and generation tasks, and the other one which implements the visualization tasks, and is able to export the generated models to a standard format, to be able to use the generated models in other applications.

Capítol 1

Introducció

1.1 Formulació del problema

La generació d'entorns grans amb gran quantitat de models, diferents entre si i amb un nombre de polígons elevat, és un problema al qual s'enfronten per exemple les companyies cinematogràfiques i les dedicades als jocs 3D. Com per exemple la creació i modelatge d'una ciutat on els edificis han de ser semblants però no idèntics, tractem amb un entorn que requereix amb una gran quantitat de models diferents els quals volem que siguin el més detallats possibles.

La creació i el modelatge d'aquest tipus d'entorns és molt costosa, ja que s'ha de delegar aquest procés a un equip d'artistes exclusivament dedicats a la creació de l'entorn, aquest procés creatiu, pot requerir fins hi tot diversos anys per a ser completat, implicant una gran inversió en temps i recursos només en la creació de l'escenari del joc o la pel·lícula. En aquest proces s'haurà de tindre en compte la mida de l'escenari i la qualitat que desitgem que tingui aquest, ja que els recursos necessaris augmenten dràsticament a mida que augmentem mida i qualitat de l'obra, doncs s'haurà de buscar un compromís entre mida i qualitat que no faci el projecte inviable.

Aquest projecte tracta sobre com emprant algorismes procedurals basats en l'ús de gramàtiques, les quals defineixen mitjançant regles per dividir, transformar i substituir un volum bàsic amb l'objectiu d'obtindre models més complexos. Mitjançant l'aplicació d'aquestes regles successivament, s'anirà refinant aquest volum bàsic fins a aconseguir generar el model més complex, amb l'objectiu d'obtindre un model amb una forma semblant a la d'un edifici.

1.2 Enfoc del problema

Al abordar el problema de la generació de models tridimensionals d'edificis, mitjançant tècniques procedurals basades en regles, trobem amb quatre grans subproblemes, el primer d'ells és la tria d'una gramàtica prou complerta com per satisfer les nostres necessitats, el segon és com parsejar aquesta gramàtica, el tercer és com generar paraules a partir de la gramàtica, i finalment com per mitjà d'aquestes paraules obtindre models tridimensionals en un format capaç de ser usat en altres programes.

Per tal de resoldre els problemes descrits anteriorment, s'ha decidit dividir el projecte en dos fases principals, tria de la gramàtica i implementació.

En la fase de tria de la gramàtica s'haurà d'escollir una gramàtica prou complerta per generar models i generar exemples d'us que demostrin el poder de la gramàtica.

L'etapa d'implementació s'ha dividit en tres gran parts: parser, generador i visualitzador/exportador. Cada una d'elles encarregada de solucionar un dels problemes descrits anteriorment.

1.2.1 Gramàtica

Usarem la gramàtica de formes(shape gramar) descrita en el paper *Procedural Modeling of Buildings*[1]. El conjunt de regles que composen la gramàtica segueixen un patró comú: $id_{regla} : S_{predecessor} : C_{aplic} \rightarrow S_{succesor} : p$ que ens expressa: si tenim el símbol $S_{predecessor}$ i és compleix la condició C_{aplic} substituirem $S_{predecessor}$ per $S_{succesor}$ amb probabilitat p, on $S_{succesor}$ pot ser un o mes símbols de la gramàtica.

1.2.2 Parser

El parser ha de ser capaç de llegir un conjunt de regles, realitzar l'anàlisi sintàctic del conjunt per verificar que pertanyen a la gramàtica, i finalment crear l'AST(Abstract Sintax Tree) a partir d'elles, aquest AST serà proporcionat al generador perquè comenci a generar els models.

1.2.3 Generador

El generador rebrà l'AST generat pel parser, i les dimensions del volum inicial, partint d'aquest volum inicial, el generador anirà construint un arbre, que posteriorment serà el model, aplicant les regles representades en l'AST. El generador aplicarà les regles sobre aquest model inicial tot seguint un ordre de prioritat, preestablert per l'usuari que ha creat el conjunt de regles, finalment quan el generador ja no pugui aplicar més regles sobre l'arbre de símbols i només quedin nodes terminals, és proporcionarà aquest arbre al visualitzador.

1.2.4 Visualitzador/Exportador

El visualitzador rebrà l'arbre de símbols generat pel generador, en forma d'string, que reconstruirà en forma d'arbre. Finalment amb el visualitzador es podrà visualitzar el model generat resultant de l'aplicació de les regles i addicionalment oferirà la possibilitat exportar-lo en format *.ply* [2].

Capítol 2

Competències tècniques

Durant la realització d'aquest projecte s'han treballat i desenvolupat diverses competències tècniques relacionades amb el projecte.

2.1 CCO1.1

Avaluar la complexitat computacional d'un problema, conèixer estratègies algorísmiques que puguin dur a la seva resolució, i recomanar, desenvolupar i implementar la que garanteixi el millor rendiment d'acord amb els requisits establerts.

Durant tota la realització del projecte s'ha intentat fer us d'estructures de dades i patrons de disseny que permetessin una major eficiència per tal d'evitar els colls d'ampolla durant el proces de generació i visualització.

En la part del Generador, caldria destacar que cada regla s'ha indexat en mapes de hash per tindre un temps d'accés constant i no recorre cada vegada el conjunt de les regles candidates de ser aplicades, s'han aplicat estratègies similars per gestionar els símbols no terminals per intentar evitar cerques innecessàries.

Per al visualitzador s'ha usat el patró singleton, per implementar el magatzem de models i evitar lectures al sistema de fitxers innecessàries.

2.2 CCO1.2

Demostrar coneixement dels fonaments teòrics dels llenguatges de programació i les tècniques de processament lèxic, sintàctic i semàntic associades, i saber aplicar-les per a la creació, el disseny i el processament de llenguatges.

Aquesta competència s'ha treballat en profunditat alhora de crear la gramàtica per generar els models. Partint d'uns predicats inicials bàsics s'han derivat la resta de predicats que formen el llenguatge, i aixi afegir complexitat i alhora crear una gramàtica molt canviable i tolerant a canvis, tenint en ment futures ampliacions de la mateixa.

2.3 CCO2.6

Dissenyar i implementar aplicacions gràfiques, de realitat virtual, de realitat augmentada i videojocs.

Tot i el projecte estar més orientat a la generació de contingut per al tipus d'aplicacions esmentades anteriorment. Aquesta competència s'ha treballat al realitzar el visualitzador, una eina que ens ofereix la possibilitat d'explorar i visualitzar una escena tridimensional, que conté el model generat, i per sobre d'això que ens permet exportar aquest model en un format estandarditzat per ser usat en tot tipus d'aplicacions amb contingut tridimensional.

Capítol 3

Conceptes bàsics

En aquest capítol es pretén introduir alguns dels conceptes bàsics en els quals es basa aquest projecte, s'introduirà el concepte generació procedural amb un breu estat de l'art, es definirà el concepte de gramàtica i s'exposaran algunes de les gramàtiques usades en la generació procedural, juntament amb la usada en aquest projecte, s'exposarà la representació dels models generats per la gramàtica i com obtindre els models tridimensionals a partir d'ells, i finalment s'introduirà el format(*ply*) en el qual s'exporten els models.

3.1 Generació procedural

La generació procedural és aquell camp que estudia algorismes que permeten generar o modificar(afegir detall) contingut automàticament, sense la intervenció d'un usuari[3]. Una de les característiques mes importants de les tècniques procedurals és l'abstracció[3], més que explicitar l'objecte que és vol generar, el que s'intenta o es pretén, es capturar la seva essència i abstraure un algorisme que sigui capaç de generar l'objecte, això en permet controlar certes característiques mitjançant paràmetres, i com a conseqüència en resulta una gran flexibilitat alhora de generar contingut diferent a partir d'un mateix patró.

Les tècniques més utilitzades avui dia per generar terrenys o textures són les tècniques basades en l'us de soroll[4], en el cas de terrenys és construirà una textura, interpretada com a mapa d'alçades partir d'aplicar cert soroll sobre una textura plana, en el cas de voler generar textures, el soroll representarà els diferents colors de la superfície. Com a tècniques mes importants hi ha

Perlin noise[5].

Depenent del tipus d'objecte que és vol generar s'usaran diferents estratègies. Per exemple si es vol generar elements naturals, com vegetació és poden seguir enfocs més naturalistes basats en el creixement de la vegetació, com per exemple arbres [6], i com van ocupant espai a mesura que es van desenvolupant. Tot i que també s'usen aproximacions basades en regles, que intenten descriure com és desenvolupa la vegetació al llarg del temps(*Sistemes-L*). Per a la generació d'edificis és prefereix seguir estratègies basades en l'us de gramàtiques de forma(*Shape Grammars*)[1] o usar estratègies de composició de models arbitràriament generats a diferents alçades [7].

3.2 Gramàtiques

Utilitzem les gramàtiques per definir un llenguatge, en el cas de la generació procedural usem les gramàtiques per definir un llenguatge les paraules del qual puguin ser interpretades com allò que volem generar.

Definirem una gramàtica[8] com a un quàdruple (Σ, V, S, P) on Σ és un conjunt no buit de símbols terminals anomenat *alfabet terminal*, V és un conjunt no buit d'elements que no pertanyen a Σ i s'anomenen *variables*, S és un símbol pertanyent al conjunt V anomenat *symbol inicial* finalment P és un conjunt finit de regles de producció de la forma $\alpha \to \beta$.

S'entén que una paraula pertany al llenguatge definit per la gramàtica si pertany al conjunt de símbols/paraules que s'obtenen d'aplicar successivament totes les regles de producció començant des del símbol inicial.

3.2.1 Sistemes-L

El sistemes-L són un tipus de gramàtiques introduïdes en 1968 per *Aristid Lindenmayer*. Es basen en intentar descriure el creixement dels sistemes al llarg del temps, la qual cosa les fa molt adequades per generar vegetació.

La generació de paraules usant aquestes gramàtiques és caracteritza, per ser un proces en el qual no és te en compte cap tipus de prioritat alhora d'aplicar les regles, sinó que a cada iteració s'apliquen paral·lelament totes les regles de producció aplicables sobre el conjunt de variables existent. Habitualment el s'aplica un nombre finit de vegades el proces d'aplicació de regles, fruit d'aquesta peculiaritat el model final podrà contindre nodes no-terminals i serà independent de l'ordre amb el qual apliquem les regles.

Usualment les regles en els sistemes-L descriuen transformacions que s'han d'aplicar, com per exemple translacions o escalats per obtindre finalment el model, o el contingut desitjat.

Tot i ser gramàtiques adequades per capturar el creixement al llarg del temps, no són sistemes adequats per caracteritzar estructures rígides i on és necessita seguir un patró estructural molt marcat, degut a la seva falta de noció de l'ordre. Per aquest motiu no s'han usat sistemes-L per a la generació procedural d'edificis en aquest projecte.

3.2.2 Gramàtiques de forma

Les gramàtiques de forma són un tipus de gramàtiques introduïdes en 1972 per *George Stiny* i en *James Gipsque*[9], basades en descriure com una forma o les seves parts poden ser transformades i/o substituïdes, amb l'objectiu d'intentar generar formes més complexes.

La generació de paraules usant aquestes gramàtiques, s'efectua aplicant seqüencialment les regles de producció sobre el símbol inicial i els seus successors, en un ordre preestablert en la gramàtica, aplicant una única regla per iteració, al contrari que els sistemes-L que les apliquen totes de cop. Les regles en quest tipus de gramàtiques descriuen com substituir una forma per una altra diferent.

Són gramàtiques adequades per descriure i generar estructures ordenades, ja que permeten guiar el proces de la formació de l'estructura degut a la seva noció de prioritat alhora d'aplicar les regles de producció sobre les variables.

3.2.3 CGA Shape grammar

És una gramàtica proposada per *Peter Wonka* i *Pascal Müller*[1], específicament dissenyada per a la generació procedural d'edificis. És una gramàtica que al igual que les gramàtiques de forma considera formes com a símbols i posseeix noció d'orde d'aplicació de les regles de producció, però mentre que una gramàtica de forma substitueix formes per altres formes, aquesta gramàtica al igual que els sistemes-L descriu les transformacions que s'han d'aplicar sobre les formes, substituint una forma/símbol per un conjunt de transformacions i símbols.

És una gramàtica que treballa sobre una configuració de formes. Una forma consisteix en un símbol i un conjunt de propietats geomètriques. El símbol identifica la forma, podent ser aquest un símbol terminal o un símbol no terminal, això permet classificar les diferents formes com a formes terminals o no terminals. Les propietats geomètriques de la forma podem variar depenent de l'estratègia que és pregui al assignar la responsabilitat sobre qui emmagatzema les transformacions aplicades a la forma.

La responsabilitat d'emmagatzemar les transformacions que s'apliquen sobre la forma, pot ser assignada a les formes, encabint les transformacions dins del paràmetres geomètrics de la forma, o es pot delegar a l'estructura que anem creant a l'aplicar les regles de producció sobre els símbols, i que posteriorment serà el model, llavors el model contindrà nodes de tipus símbol i de tipus transformació. En aquest projecte s'ha optat per delegar la responsabilitat d'emmagatzemar les transformacions al model, ja que s'ha considerat un mètode més natural, encara que si és vol treballar amb el menor tipus de nodes possible, i ser més eficient, és preferible assignar aquesta responsabilitat a les formes.

Les regles de producció de la gramàtica és definiran de la següent manera:

$$id_{regla} : S_{predecessor} : C_{aplic} \rightarrow S_{succesor} : p$$

On id_{regla} és un identificador únic de la regla, $S_{predecessor}$ és un el símbol que identifica la forma que serà substituïda per $S_{succesor}$, que pot ser un símbol a un conjunt de símbols i/o transformacions, C_{aplic} serà una expressió lògica que s'haurà d'avaluar a cert per poder aplicar la regla i finalment p és la probabilitat de que la regla s'apliqui.

És poden classificar els diferents tipus regles de producció que posseeix la gramàtica en quatre grans famílies segons l'objectiu que persegueixen.

1. Regles d'àmbit o de substitució simple, són regles que substitueixen el símbol $S_{predecessor}$ per el conjunt de símbols i/o transformacions que conté $S_{succesor}$.
2. Regles de separació, són regles que expressen com subdividir una forma segons un eix establert, permeten subdividir la forma en formes de mida fixe o mida variable segons la mida que s'assigni a la forma $S_{predecessor}$.
3. Regles de repetició, són regles que busquen subdividir una forma en un determinat símbol, de mida fixe, sobre un o varis eixos de la forma $S_{predecessor}$.
4. Regles separació de components, són regles que permeten dividir una forma en el seus components de menys dimensions, per exemple la descomposició d'un cub en les seves cares, pas de tres dimensions a dues dimensions.

3.3 Representació del models

La representació o codificació dels models està estretament lligada al tipus de gramàtica que s'ha volgut usar per generar-los. Concretament en el nostre cas el model serà l'arbre creat per la gramàtica, resultat d'aplicar les regles de producció a partir del símbol inicial fins que no quedi cap símbol no terminal.

El model resultant serà un arbre que posseirà nodes de tipus símbol terminals a les fulles i no terminals entre l'arrel i els fulles podent ser aquests, nodes amb símbols auxiliars o nodes que expressen una transformació. Per obtindre el model generat a partir d'aquest arbre, s'haurà de recórrer aquest arbre des de l'arrel a les fulles aplicant les transformacions que anem trobant pel camí als nodes fills d'aquestes, fins arribar als nodes terminals.

Els nodes terminals ens descriuran quina forma s'haurà de renderitzar, sent aquesta composició de renderitzacions, de nodes terminals als quals apliquem una serie de transformacions, el que finalment constituirà el model/edifici generat per la gramàtica.

3.4 Format ply

El format *.ply* [2] és el format escollit per exportar els models generats i aixi permetre la possibilitat d'usar aquests models en altres aplicacions, ja que el format *.ply* és un format estandarditzat. EL format *.ply* ha estat escollit degut a que és un format fàcil d'entendre, és senzill codificar models en quest format, i es suportat per quasi totes les aplicacions de modelatge i totes les llibreries de lectura de models tridimensionals.

L'estructura típica d'un fitxer *.ply* es divideix en una capçalera, una llista de vèrtexs i una llista de cares, com a l'exemple de la Figura 3.1.

En la capçalera consta de la informació que hi ha des de la paraula `ply` fins a la paraula clau `end_header`, en la capçalera s'indica el nombre vertex que posseeix el model i les propietats que s'associarà a cada vertex, com a mínim les coordenades dels vertex, desprès s'indica el nombre de cares del model.

A continuació de la paraula clau `end_header` s'introdueixen la llista de vertex de l'objecte amb totes les propietats definides en la capçalera, tot just desprès de la llista de vertex s'introduirà la llista de les cares que formen l'objecte, per cada cara s'introduirà el nombre de vertex que la formen i a continuació els index dels vertex que la formen

```
ply
format ascii 1.0
element vertex 8
property float x
property float y
property float z
element face 6
property list uint8 int32 vertex_index
end_header
0 0 0
0 0 1
0 1 1
0 1 0
1 0 0
1 0 1
1 1 1
1 1 0
4 0 1 2 3
4 7 6 5 4
4 0 4 5 1
4 1 5 6 2
4 2 6 7 3
4 3 7 4 0
```

Figura 3.1: Fitxer en format ply que codifica un cub unitari

Capítol 4

Solució proposada

En aquest capítol s'explicarà la solució al problema de la generació procedural d'edificis que es proposa en aquest projecte usant la gramàtica CGA shape. S'explicara el funcionament del llenguatge creat en aquest projecte que permet definir gramàtiques, i es descriuran les diferents parts software(parser, generador, visualitzador) implementades.

Inicialment es volia implementar el projecte com una única aplicació escrita en *C++* , ja que *C++* permet treballar amb el gestor de finestres Qt i la API OpenGL per poder crear la nostra interfície i la finestra on visualitzarem els models creats, i a més a més *C++* també és compatible amb el generador de parsers LL(1) antlr[10], però degut a la problemes al intentar generar el parser amb antlr per als llenguatges *C++* o *C* i a causa de la poca documentació de la qual disposa antlr referent a *C* i *C++* , finalment es va optar per implementar el parser i el generador en *Java*, i implementar el visualitzador en *C++* . Això comporta implementar dos sistemes separats que han de comunicar-se, un sistema serà el parser més el generador, i l'altre sistema serà el de visualització.

El visualitzador serà l'encarregat d'invocar al conjunt del parser i el generador quan calgui, establint la comunicació per linea de comandes i obtenint el model generat pel sistema parser més generador, del sistema de fitxers del sistema operatiu.

4.1 El llenguatge

S'ha creat un llenguatge simple, que permet especificar gramàtiques CGA shape. Aquest llenguatge s'ha especificat amb antlr, el qual ens proporcionarà el parser i ens construirà l'AST tal i com s'ha especificat alhora de definir el llenguatge.

Veiem un exemple del llenguatge que defineix una gramàtica per crear blocs d'oficines Figura 4.1, més endavant s'explicarà en detall el funcionament de l'exemple un cop introduïdes totes les regles.

```
/*Definició de variables*/
VARS:
type 1
//Definició de prioritat
PRIORITY 1:
1 lot --> Subdiv(Z, z_dim*rand(0.3, 0.5), 1r) {facade| sidewings}
2 sidewings
        --> Subdiv(X, x_dim*rand(0.2, 0.6), 1r){sidewing|}:0.5
        --> Subdiv(X, 1r, x_dim*rand(0.2, 0.6)){|sidewing}:0.5
3 sidewing
        --> S(x_dim, y_dim, z_dim*rand(0.4, 1)) facade : 0.5
        --> S(x_dim, y_dim*rand(0.2, 0.9), z_dim*rand(0.4, 1))
            facade : 0.3
        --> : 0.2
4 facade : (type == 1) --> I("cube")
5 facade : (type == 2) --> I("cylinder")
```

Figura 4.1: Gramàtica d'exemple usada per crear blocs de pisos

Podem veure clarament dos grans seccions, una secció precedida per la paraula *VARS* on es poden declarar variables per usar-les posteriorment en les regles, i una secció precedida per la paraula *PRIORITY* on es declaren les regles de producció de la gramàtica. Tota gramàtica especificada ha de tindre com a mínim una secció *PRIORITY* mentre que la secció *VARS* és opcional.

També és pot observar que al igual que el el llenguatge *C++* , el llenguatge dissenyat ofereix la possibilitat d'utilitzar comentaris per bloc amb */* */* i comentaris per línia amb *//*.

4.1.1 Tipus de dades

Aquest llenguatge només posseeix dos tipus de dades, boleans i decimals. Els boleans només són usats en l'avaluació de condicions, mentre que els decimals són el principal tipus de dades amb les que treballa el llenguatge, els usos principals dels decimals són: ser assignats a variables en la secció *VARS*, usar decimals i/o variables per especificar els paràmetres de les transformacions, definir la probabilitat d'aplicació de la regla. El llenguatge a més a més ofereix la possibilitat d'obtindre decimals de forma aleatòria dins d'un rang amb la sentencia $rand(min, max)$

4.1.2 Prioritat

El llenguatge ofereix dos nivells de prioritat, la prioritat entre blocs *PRIORITY*, i la prioritat entre les regles dins d'aquest blocs.

La prioritat entre blocs la defineix l'ordre en que són declarats, addicionalment als blocs se'ls hi assigna un numero per identificar-los, que usualment concorda amb l'ordre en el que són declarats.

La prioritat entre les regles dins del blocs de prioritat, es defineix assignant un identificador enter únic dins del bloc, entre u i el numero de regles que pertanyin al bloc. Finalment la prioritat dins del bloc s'obtindrà ordenant les regles de menor a major segons el seu identificador, sent l'identificador amb menor valor el de màxima prioritat.

4.1.3 Regles

Partint de com es defineix una regla en aquest llenguatge, s'explicaran els diferents elements dels quals disposa el llenguatge per definir regles.

$$id_{regla} \quad S_{predecessor}(: C_{aplic})?\texttt{-->}S_{succesor}(: p)?$$

On id_{regla} és l'identificador únic dins del bloc de prioritat on és troba la regla i que marca la prioritat d'aquesta dins del bloc, $S_{predecessor}$ és el símbol sobre el que la regla s'aplicarà i serà substituït per $S_{succesor}$, C_{aplic} és una expressió booleana que s'ha d'avaluar a cert per poder aplicar la regla. p és la probabilitat que la regla s'apliqui. $S_{succesor}$ és el símbol o símbols pels quals $S_{predecessor}$ serà substituït, segons el tipus de $S_{succesor}$ que tinguem podem parlar de diferents tipus de regles. Els ()? són per emfatitzar que els elements C_{aplic} i p són opcionals alhora de declarar una regla, i si no són declarats explícitament C_{aplic} s'avalua a true sempre i p a u.

```
1 A --> T(0,0,3.5)S(8,0.5,4) B
2 B --> [Rx(45)I("cube")]T(0,0,2) C
3 C : (rand(0,1) <= 0.5) --> D
4 C --> I("door.obj") : 0.8
```

Figura 4.2: Diferents exemples de regles definides amb el llenguatge

També podrem definir regles aniuades, que són regles amb múltiples $S_{succesor}$ i amb una probabilitat diferent que s'apliquin aquest diferents successors, les diferents probabilitats assignades a cada successor hauran de sumar sempre u, tal i com es pot comprovar en l'exemple de la Figura 4.3.

```
3 sidewing
        --> S(x_dim, y_dim, z_dim*rand(0.4, 1)) facade : 0.5
        --> S(x_dim, y_dim*rand(0.2, 0.9), z_dim*rand(0.4, 1))
                facade : 0.3
        --> : 0.2
```

Figura 4.3: Regla amb probabilitat aniuada

4.1.3.1 Expressions

Existeixen dos tipus d'expressions, les expressions booleanes que principalment són usades en la definició de les condicions d'aplicabilitat de les regles i les expressions numèriques usades en les condicions d'aplicabilitat, si aquestes requereixen càlculs algebraics, i per realitzar càlculs numèrics al definir els atributs de les transformacions.

En les expressions es poden usar variables, nombres aleatoris, nombres decimals , booleans, a més a més es poden usar variables especials que fan referencia a les dimensions del símbol sobre el qual s'aplica la regla. Aquests símbols són: `x_dim`, `y_dim` i `z_dim`. Que respectivament fan referencia a la dimensió en x, y i z del símbol sobre el que s'aplica la regla.

Alhora de d'avaluar les expressions es realitzara la comprovació de tipus per tal de no aplicar operacions aritmètiques entre enters i booleans o efectuar operacions booleanes entre tipus de dades diferents.

4.1.3.2 Regles de substitució

Les regles de substitució són regles en les quals és modifiquen les formes mitjançant transformacions. Podem descriure tres tipus diferents de transformacions: $T(t_x, t_y, t_z)$ especifica el vector de translació que s'aplicarà a les formes, $R_x(\alpha)$, $R_y(\alpha)$, i $R_z(\alpha)$ especifiquen la rotació l'angle α de rotació i l'eix de rotació sobre el qual és rotaran les formes, i $S(s_x, s_y, s_z)$ és el vector que especificarà la mida de les formes, cada component sent la mida que prendrà la forma en x, y i z respectivament.

Utilitzarem claudàtors, *[* i *]* per apilar i desapilar l'àmbit actual, això serà útil per aïllar transformacions en una regla que no afectaran a tots els símbols que genera la regla, ja que tots els símbols generats per la regla seran afectats per les transformacions del seu mateix àmbit.

Per afegir primitives geomètriques s'usarà la comanda `I("idobjecte")`, que ens afegira un node terminal a l'arbre de generació que tindrà associat la primitiva identificada per *idobjecte*. Típicament s'afegiran primitives simples de tipus cub o cilindre, però amb aquesta comanda podem importar models tridimensionals, fent que *idobjecte* sigui el nom complert de l'objecte que es vol importar per exemple, `I("columna.ply")`.

```
1 A --> [T(0,0,6)S(8,10,18)I("cube")] B
2 B --> T(6,0,0)S(7,13,18)I("cube") C
3 C --> T(0,0,16)S(8,15,8)I("cylinder")
```

Figura 4.4: Exemple de regla de substitució

Com es pot veure a l'exemple de la Figura 4.4 s'utilitzen els claudàtors per aïllar les dos formes terminals de les regles (2) i (3) de les transformacions, a les quals és sotmesa la forma terminal de la regla (1).

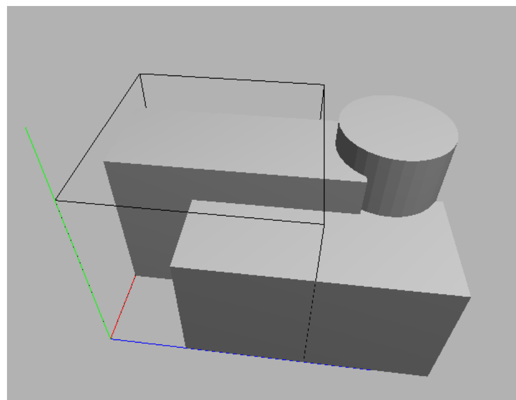

Figura 4.5: Model generat usant la gràmatica descrita en la Figura 4.4.

Com es pot observar en l'exemple de la Figura 4.5, al no usar cap mida en relació al volum inicial, el model generat sobresurt de les dimensions del volum inicial. També es pot observar que els símbols definits per les regles (2) i (3) no han estat afectats per les transformacions efectuades en la regla(1).

4.1.3.3 Regles de separació

Les regles de separació són aquelles que ens descriuen com dividir una forma, en altres formes, es caracteritzen per que el seu $S_{succesor}$ és de la forma: $Subdiv(eix, divisions)formes$.

Els $S_{succesor}$ de les regles de separació s'identifiquen amb la paraula clau `Subdiv` que entre parèntesis te una serie de paràmetres. Primer de tot defineix sobre quin eix es farà la subdivisió utilitzant les lletres majúscules X, Y i Z cada una d'elles indicant la subdivisió en l'eix del mateix nom. Després s'especifica la mida de les subdivisions que és realitzaran a la forma $S_{predecessor}$, i finalment entre claus es definirà quina forma o conjunt d'elles s'ha de posar en cada divisió.

La mida de les subdivisions es pot especificar de forma absoluta, amb nombres reals, o de forma relativa a l'objecte. És necessari tindre la possibilitat de definir particions relatives a la mida de l'objecte, ja que permet crear regles de separació que es poden adaptar a objectes de mides diferents, mentre que si no fos permès s'haurien de crear regles de separació per cada mida possible de les formes amb les que tractem. Els valors que s'usen per definir les mides és consideren valors absoluts per defecte, i per denotar valors relatius s'utilitza la lletra r.

```
1 A --> Subdiv(X, x_dim*rand(0.2, 0.6), 1r, 2r){B||C}
2 B --> I("cube")
3 C --> I("pyramid.ply")
```

Figura 4.6: Exemple de regla de separació

Per obtindre les dimensions assignades als valors relatius com per exemple els valors relatiu de la Figura 4.6, per cada valor relatiu r_i s'aplicarà la formula següent $r_i * (dim - \sum abs_i)/\sum r_i$, on dim serà la mida total de la forma $S_{predecessor}$ en l'eix que es fa la divisió, $\sum abs_i$ serà la suma de tots els valors absoluts, i $\sum r_i$ la suma de tots els valors relatius.

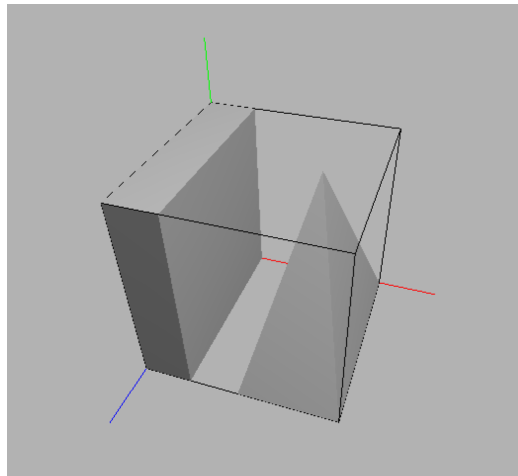

Figura 4.7: Model generat usant la gràmatica descrita en la Figura 4.6.

En l'exemple de la Figura 4.7, es pot veure clarament com s'ha dividit el volum inicial en tres, sobre el seu eix X, i entre els dos models que hi apareixen hi ha un buit el qual la seva mida en X és exactament la meitat que la de la piramide, tal com indica la regla (1) de la Figura 4.6.

4.1.3.4 Regles de repetició

Les regles de repetició són una generalització de les regles de separació, que serveixen per dividir una forma en sub formes de mida fixe, permetent dividir segons els eixos basics X, Y, Z o composició d'aquests, el seu $S_{succesor}$ és de la forma: $Repeat(eix, mida)formes$.

```
1 A --> Repeat(XYZ, 3, 3, 3){B}
2 B --> I("monkey.ply")
```

Figura 4.8: Exemple de regla de repetició

En lexemple anterior Figura 4.8, la forma A serà subdividida en el seus plans XYZ per el màxim nombre de formes B amb dimensió X igual a tres, dimensió Y igual a tres i dimensió Z igual a tres que s'hi puguin encabir, $repeticions_x = repeticions_y = repeticions_z = \lceil x_dim/3 \rceil$.

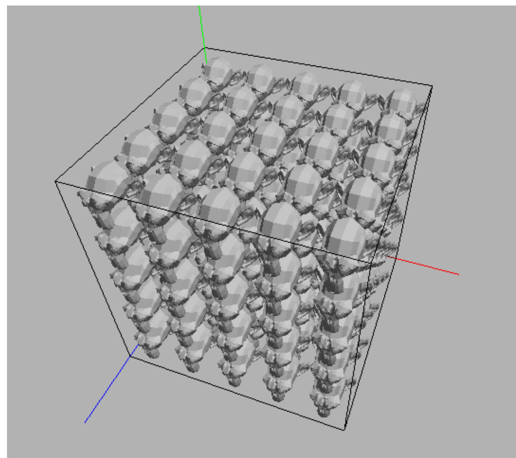

Figura 4.9: Model generat usant la gràmatica descrita en la Figura 4.8.

En l'exemple de la Figura 4.9, es pot veure que s'ha dividit el model inicial en varis models *monkey*, en aquest cas el model inicial era de 15x15x15 i per això ha estat dividit en 125 *monkey*.

4.1.3.5 Regles de separació de components

Les regles de separació de components ens permeten separar un volum bàsic(cub) en els seus components, cares, arestes i vertex, fent possible passar d'una forma de tres dimensions a una forma de dos dimensions o d'una dimensió. Les regles de separació de components són de a forma:$Comp(tipus, var)\{forma\}$

On *tipus* identifica quin tipus de component és seleccionarà, el valor de *var* permetrà seleccionar si n'hi ha més d'un component del tipus *tipus*, quin d'ells es vol seleccionar, i finalment *forma* serà l'identificador de la forma per la que substituirem els components seleccionats.

Per codificar formes de menys de tres dimensions, és modificarà el vector $S(s_x, s_y, s_z)$ de les formes, posant el valor referent als eixos que no tenen dimensió a zero. Per tornar a transformar els objectes de menys dimensions a objectes tridimensionals caldrà aplicar una transformació d'escalat, tornant a posar els valors del vector $S(s_x, s_y, s_z)$ que estan a zero, a valors majors que zero.

```
1 A --> Comp(sidefaces){B}
2 B --> S(1, y_dim, z_dim) I("cube")
```

Figura 4.10: Exemple de regla de separació de components

En l'exemple anterior Figura 4.10, la regla (1) crearà una forma B per cada cara lateral de la forma A, aleshores la regla (2) tornarà a donar dimensió X a les cares i tornaràn a ser formes tridimensionals, tal com es pot veure a la Figura 4.11.

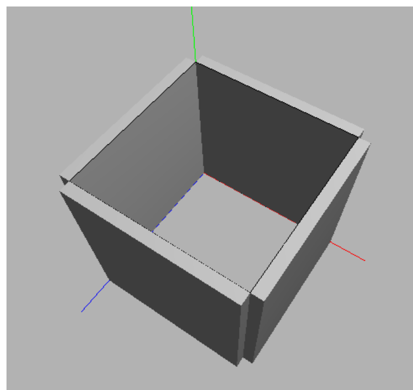

Figura 4.11: Model generat usant la gràmatica descrita en la Figura 4.10.

4.1.4 Exemple de generació

En l'exemple mostrat en la Figura 4.1, es mostra un gramàtica que genera models en forma de blocs de pisos. Aquesta gramàtica partira el model inicial, identificat amb el simbol `lot`, en dos parts per l'eix Z, una de les parts decidirà la mida de l'altre, ja que la part absoluta `z_dim*rand(0.3, .05)` tindrà una mida aleatòria en cada generació i el valor relatiu `1r` es limitarà a agafar la resta de l'espai lliure. A la partició assignada al símbol `sidewings`, se li aplicarà la regla (2) que és una regla amb probabilitat aniuada, que substituirà el símbol `sidewings` per un espai buit i un símbol `sidewing`, amb el 50% de probabilitat que l'espai buit es trobi a la dreta o a l'esquerre del símbol `sidewing`, posteriorment s'aplicarà la regla(3) sobre el simbol `sidewing`, que idènticament a la regla (2) és una regla amb probabilitat aniuada, amb probabilitat del 50% es modificarà la dimensió en l'eix Z de la forma i és substituirà per el símbol `facade`, amb una probabilitat del 30% es modificaran els eixos Z i Y de la forma i és substituirà per el símbol `facade`, i amb un 20% de probabilitat no es substituirà per res, creant un espai buit. Finalment els símbols `facade` es substituiran per cubs o cilindres segons el paràmetre `type`. Es pot veure un exemple dels models generats per aquest conjunt de regles, en la Figura 5.5.

Capítol 5

Implementació

En aquest capítol s'explicarà el funcionament i l'estructura dels sistemes implementats. Com s'ha dit anteriorment, s'han implementat dos sistemes diferents, el conjunt de l'interpret més el generador, i el visualitzador, cada un escrit en un llenguatge de programació diferent *Java* i *C++* respectivament.

5.1 Parser més Generador

El parser i el generador, formen un sol bloc implementat en *Java*, i són els encarregats de llegir una gramàtica, i posteriorment generar els models resultants d'aplicar aquesta gramàtica. El bloc que formen el parser més el generador, ha estat dissenyat com una aplicació apart anomenada ParseGen, que no necessita el visualitzador per funcionar, i pot ser usada perfectament des de la terminal.

A l'invocar l'aplicació des de la terminal, es poden afegir diversos flags per configurar les opcions de les que disposa, aquestes opcions són mostrades si s'invoca amb el flag `-help`.

```
$ ./bin/ParseGen -help
usage: ParseGen [options] file
 -ast <file>                write the AST
 -dotpdf                    dump the AST in dot and pdf format
 -gen <x_dim y_dim z_dim>   generate and define initial size
 -help                      print this message
 -model <file>              write generated model to a file
```

Figura 5.1: `help` de l'aplicació ParseGen

La opció `-ast` genera una representació textual de l'AST en format dot. Amb la opció `-dot` la representació estarà en format dot, però que podrà ser convertida a representació gràfica usant el programa `dot`, i addicionalment també generarà la representació gràfica de l'AST en format pdf. La opció `-gen` serveix per dir al programa que generi un model a partir de les regles que se li proporcionen, a partir d'un volum inicial amb les mides que s'especifiquen en el flag. La opció `-model` serveix per escriure el model generat a un fitxer, per després importar aquest model amb el visualitzador o alguna eina que entengui el format, i poder visualitzar el model generat en tres dimensions.

5.1.1 Parser

El parser s'ha implementat amb l'ajuda d'antlr. Antlr és una eina escrita en java, la qual a partir d'un fitxer *.g*, on es defineix la gramàtica d'un llenguatge i es descriu com construir l'AST, crea les classes necessàries per reconèixer el llenguatge i proporcionar l'AST d'aquest. Per poder personalitzar l'AST i afegir informació a la que proporciona antlr, s'han afegit dos classes extres amb les noves definicions dels nodes de l'AST.

El parser serà l'encarregat d'identificar els errors sintàctics que hi pugui haver en el fitxer d'entrada, i així detectar estructures no permeses en el llenguatge o errors d'escriptura per part de l'usuari que ha creat el fitxer d'entrada, i informarà en quina línia del fitxer d'entrada es troba l'error.

Finalment el parser després de processar un fitxer d'entrada escrit amb el llenguatge que hem definit per generar edificis, proporcionarà al generador una estructura de dades com la que es mostra en la Figura 5.2.

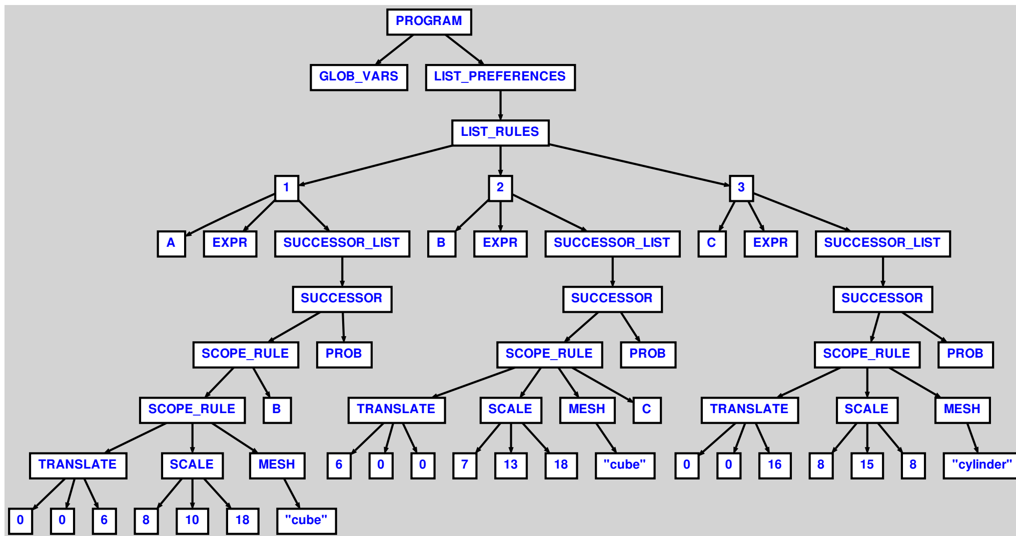

Figura 5.2: AST generat per el conjunt de regles descrites en la Figura 4.4

5.1.2 Generador

El Generador, serà l'encarregat de rebre l'AST del parser i a partir d'ell i d'un volum inicial, començar a generar el model mitjançant l'aplicació de les regles descrites en l'AST, a més a més mentre el generador va construint el model aplicant les regles de producció de la gramàtica, anirà realitzant l'anàlisi semàntic de la gramàtica que ha creat l'usuari, comprovarà errors de tipus, i construccions incoherents com regles de subdivisió amb un nombre diferent de divisions i formes que han d'ocupar les divisions. Si en algun moment durant el proces de generació encara queden símbols no-terminals i no és possible aplicar més regles, el sistema informarà amb un error que la gramàtica genera símbols no terminals sense cap regla assignada.

5.1.2.1 Generació de paraules

Per començar a generar el model es creara un símbol inicial, que tindrà com a identificador el símbol $S_{predecessor}$ més prioritari del conjunt de regles, amb les dimensions proporcionades al generador juntament amb el flag `-gen` .

EL proces de generació es pot descriure en tres fases, es parteix d'un conjunt de símbols no terminals A, que només conte el símbol inicial i és procedeix de la següent manera: *(1)* es selecciona la regla amb més prioritat, que te com a $S_{predecessor}$ un símbol pertanyent al conjunt A i es comprova si és satisfa la

probabilitat si no és aixi es selecciona la segona regla amb més prioritat que te com a $S_{predecessor}$ un simbol del conjunt A i així successivament, (2) s'aplica la regla sobre el símbol si compleix la condició C_{aplic} i obtenim el conjunt de símbols successors B a aquest conjunt no hi pertanyen ni les transformacions ni els nodes terminals, *(3)* eliminarem el símbol $S_{predecessor}$ del conjunt A i hi afegirem els símbols del conjunt B. Mentre quedin símbols en el conjunt A s'anirà repetint el pas *(1)*.

5.1.2.2 Format

El generador obtindrà un model en forma d'arbre que despès emmagatzemarà en un fitxer que s'especificarà amb el flag `-model`, que tindrà *.model* per extensió, el generador bolcarà el nodes de l'arbre en pre-ordre seguint el format següent:

$$id\ \ mod\ \ trans\ \ x\ \ y\ \ z\ \ \#fills$$

Cada línia del fitxer contindrà un node, amb *id* que serà el nom del node, en el cas dels volums geomètrics serà l'identificador del model, *mod* i *trans* ens diran del tipus de node que és tracta, si és una transformació o un node terminal, en cas de no ser cap dels dos, serà una variable auxiliar, els següents camps indicaran les dimensions de la forma, i per ultim $\#fills$ indicarà el nombre de fills que te el node en qüestió.

```
lot 0 0 10.0 20.0 10.0 2
T 0 1 0.0 0.0 0.0 1
Auxiliar 0 0 10.0 20.0 4.186853101368595 1
facade 0 0 10.0 20.0 4.186853101368595 1
cube 1 0 10.0 20.0 4.186853101368595 0
T 0 1 0.0 0.0 4.186853101368595 1
Auxiliar 0 0 10.0 20.0 5.813146898631405 1
sidewings 0 0 10.0 20.0 5.813146898631405 2
T 0 1 0.0 0.0 0.0 1
Auxiliar 0 0 2.3729426646795555 20.0 5.813146898631405 1
sidewing 0 0 2.3729426646795555 20.0 5.813146898631405 1
AuxiliarScale 0 0 2.3729426646795555 20.0 2.814121835031589 1
facade 0 0 2.3729426646795555 20.0 2.814121835031589 1
cube 1 0 2.3729426646795555 20.0 2.814121835031589 0
T 0 1 2.3729426646795555 0.0 0.0 1
Auxiliar 0 0 7.627057335320444 20.0 5.813146898631405 0
```

Figura 5.3: Exemple de model generat usant les regles descrites en la Figura 4.1

5.1.3 Organització del sistema

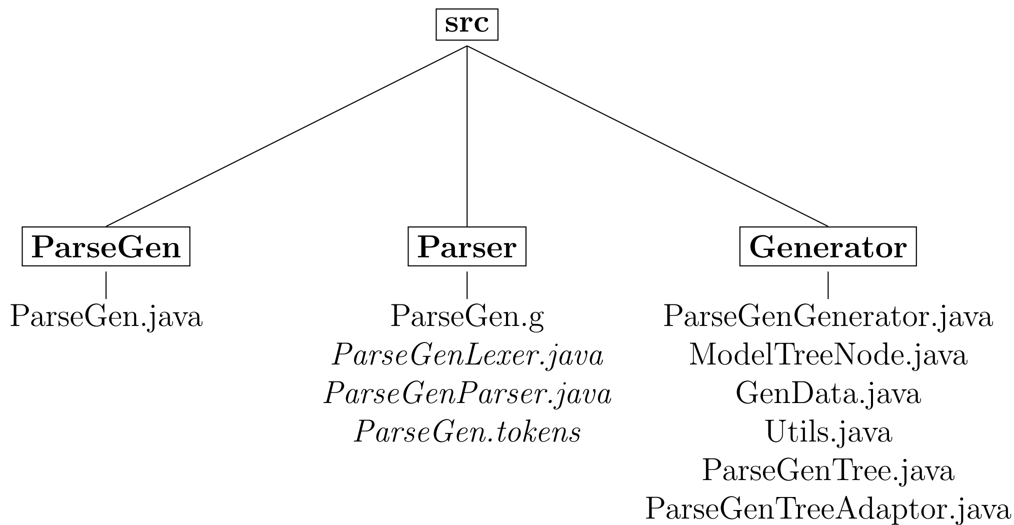

Figura 5.4: Estructura del directori `src` que conte el codi font del sistema parser més generador.

El sistema que formes en parser més el generador, està organitzat en tres paquets descrits en la Figura 5.4.

ParseGen: contè la classe ParseGen, que és la classe principal, i la que gestionarà i cridarà al parser i al generador.

Parser: conté la gramàtica i les classes encarregades de crear l'AST. Originalment el fitxer només conté l'arxiu `ParseGen.g`, els altres arxius que apareixen en cursiva a la Figura 5.4, són creats automàticament per antlr.

Generador: conte les classes necessàries pel generador organitzades en diferents arxius:

- *ParseGenGenerator.java.* Conte el nucli del generador, s'encarrega d'aplicar les regles descrites en l'AST sobre els símbols no-terminals a mesura que construeix el model.
- *ModelTreeNode.java.* Conte la classe *ModelTreeNode*, que defineix l'estructura del nodes del model, i serà l'encarregada d'emmagatzemar el model.
- *GenData.java.* Conte la classe *GenData* que s'encarrega de representar els tipus de dades diferents que tenim en la gramàtica, valors decimals i booleans, i efectua les operacions entre ells.

- *Utils.java.* Conté operacions per ajudar a computar les translacions dels objectes segons la seva rotació, útil per a les petites transformacions locals que s'apliquen en els proces de separació de components.
- *ParseGenTree.java.* Conté la subclasse de l'AST genèric d'antlr, que afegeix informació i funcionalitats als nodes de l'AST.
- *ParseGenTreeAdaptor.java.* Conte la subclasse necessaria per antlr per poder usar i crear els nodes de l'AST definit en el fitxer *ParseGen-Tree.java.*

5.2 Visualitzador

El visualitzador serà l'encarregat de gestionar la interacció amb l'usuari i oferir funcionalitats per generar, visualitzar i exportar models. El visualitzador, ha estat implementat usant el llenguatge *C++* juntament amb: *Qt* per gestionar la interacció amb l'usuari i al finestra, OpenGl per gestionar la visualització del model generat en una escena tridimensional explorable, i la llibreria *assimp* per importar i renderitzar models tridimensionals auxiliars.

5.2.1 Funcionalitats

El visualitzador ofereix diverses funcionalitats a l'usuari, i automatitza el proces de generació i visualització, només requerint de l'usuari el fitxer amb la gramàtica que es vol usar i les dimensions del model inicial.

- Carregar un fitxer definint la gramàtica: El visualitzador permetrà carregar un fitxer escrit en el llenguatge definit en la secció 4.1, que posteriorment serà usat per generar el model, els fitxer que defineixin la gramàtica, els identificarem amb l'extensió `.buildgen`.
- Generar i visualitzar un model: Quan s'hagi carregat el fitxer amb el conjunt de regles, es podrà demanar al visualitzador que cridi a l'aplicació *ParseGen*, que serà l'encarregat de generar el model a partir de la gramàtica definida en el fitxer i les dimensions definides en el visualitzador. Quan l'aplicació hagi acabat de generar el model, el visualitzador mostrarà en pantalla el model generat. Si el model Generat requereix la inclusió de models addicionals, usant la invocació en la gramàtica de `I("porta.obj")` explicat en la secció 4.1.3.2, com per exemple portes, aquests hauran d'estar en la carpeta `models/3D` que és troba juntament amb el visualitzador.
- Gestionar i mostrar els errors: El fitxer que llegim definint la gramàtica pot contindre errors, tant errors sintàctics al definir les regles com errors semàntics, que l'aplicació *ParseGen* detectarà. El visualitzador serà el responsable de capturar aquests errors i mostrar-los a l'usuari.
- Exportar els models generats: Un cop s'hagi generat un model usant la gramàtica i es pugui visualitzar, usant el visualitzador es podrà exportar aquest model a un model en format *.ply* per ser usat posteriorment en altres projectes.

- Llegir i visualitzar models: Usant el visualitzador podrem visualitzar models ja generats amb l'aplicació *ParseGen*, o que segueixin el format descrit en la secció 5.1.2.2. Addicionalment es pondrà carregar i visualitzar models en formats estàndard(*.ply*, *.obj*, *.3ds*).

5.2.2 Organització del visualitzador

EL visualitzador s'organitza sota un projecte Qt amb el mateix nom, que conte totes les classes necessàries per implementar el visualitzador, la carpeta `ParseGen` que conté la implementació de l'aplicació *ParseGen*, la carpeta `models` on han d'estar ubicats els models auxiliars que puguin ser necessaris.

Les classes necessàries per implementar el visualitzador són:

- *GLWidget*: Es la classe principal, encarregada de gestionar la finestra OpenGL, aquesta classe s'encarrega de processar els esdeveniments provocats per l'usuari i mostrar una resposta gràfica, per exemple: quan l'usuari vol rotar l'escena, clica sobre la finestra i desplaça el ratolí, la classe *GLWidget* s'encarregarà de mostrar el model rotat.
- *MainWindow*: Es la classe que gestiona la interfície gràfica, i s'encarrega d'establir la comunicació entre els esdeveniments de la interfície i les classes més internes del sistema com per exemple, *GLWidget*.
- *MeshStorer*: Una classe `singleton`, que actua com a biblioteca de models, aixi si un model generat requereix un model auxiliar moltes vegades, amb l'ajut d'aquest classe només és farà la importació del model una única vegada.
- *MeshModel*: Adaptador, encarregat de gestionar els models importats amb la llibreria *Assimp*.
- *Model*: S'encarrega de gestionar els model Generats, te les responsabilitats de llegir un model en el format anteriorment especificat i representar-lo.
- *PlyExporter*: Classe que a partir d'un conjunt de cares que formen un model, exporta aquest model en format *.ply*, per que aquest pugui ser usat en un futur en altres aplicacions i projectes.
- *CoordinateSystem*: Aquesta classe representarà un sistema de coordenades, local per a cada cara del model. Serà necessària per exportar el model i retornar les coordenades de les cares que el formen ja transformades al sistema de coordenades general del model.

- Classes auxiliars: són les classe *Point* i *Utils*, que implementen la representació de punts i funcions útils per a la realització de càlculs auxiliars, respectivament.

5.2.3 Exemples

A continuació es mostrarà la visualització del model fruit d'aplicar la gramatiac descrita en la Figura 4.1 que s'han usat com a exemple en capítols posteriors, un exemple exemple on es pot veure al visualitzador informat a l'usuari, que hi ha un error en la gramàtica i no és pot generar el model a partir d'ella, i finalment el render d'un model exportat des del visualització i renderitzat amb *blender*.

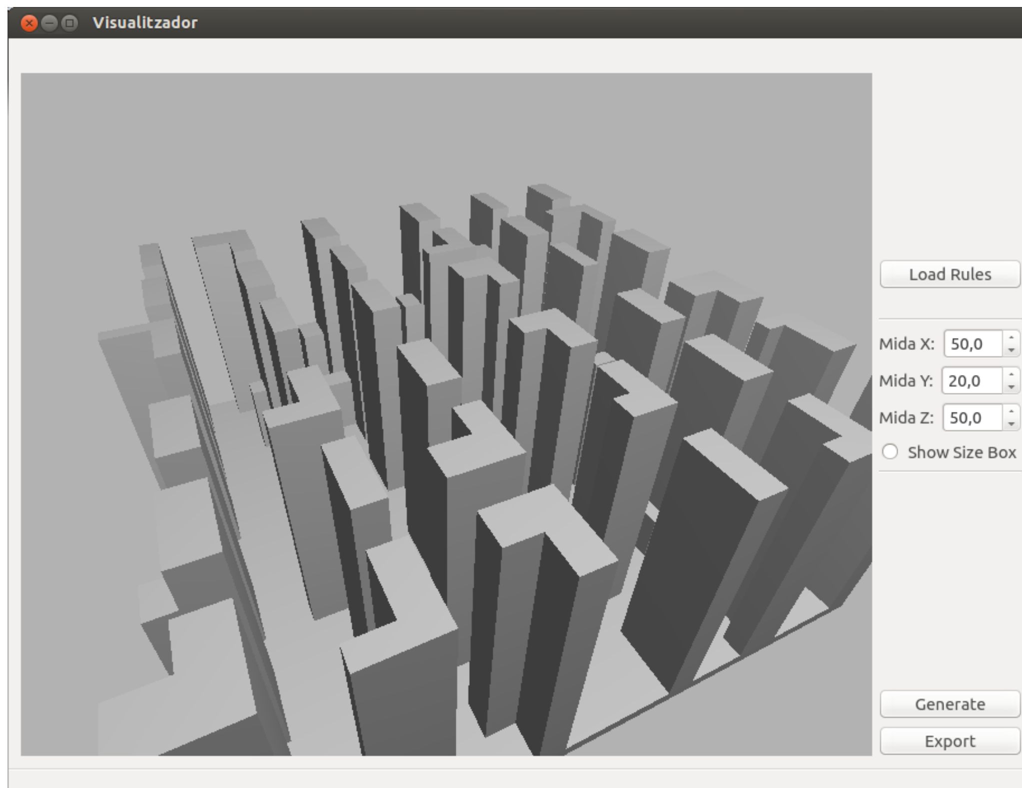

Figura 5.5: Composició de models generats usant la gràmatica descrita en la Figura 4.1

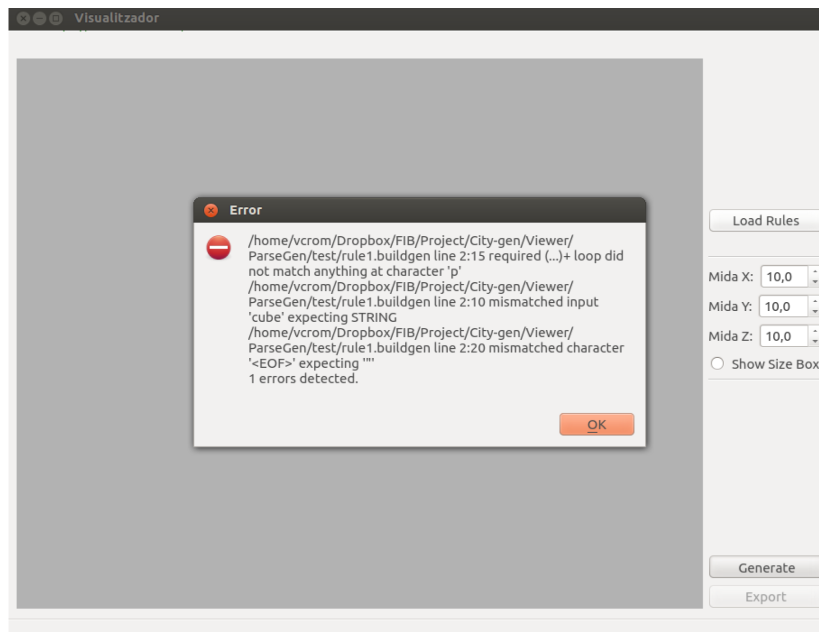

Figura 5.6: Missatge d'error obtingut intentant generar un model amb una gramàtica incomplerta que conté errors.

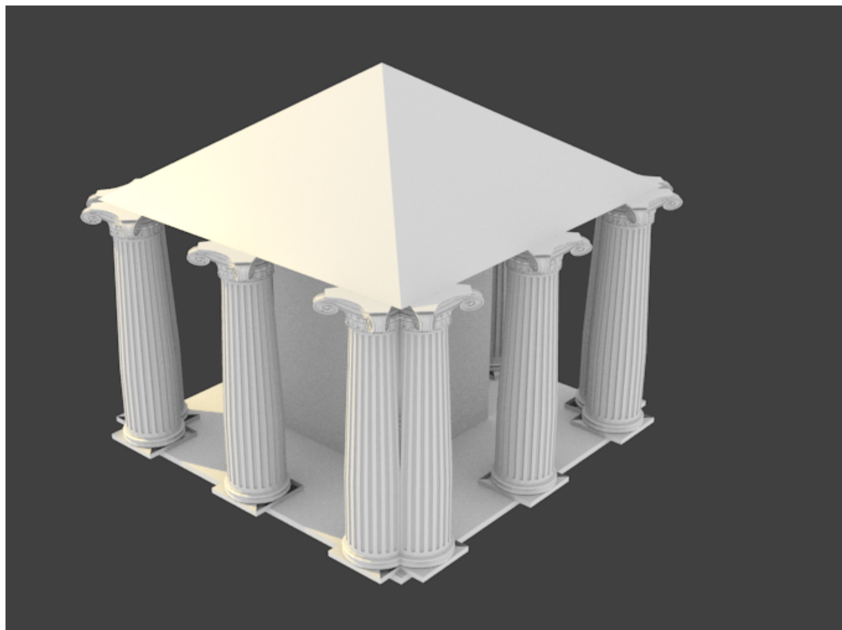

Figura 5.7: Exemple de model generat i exportat a format *.ply* amb el software dissenyat pel projecte, renderitzat amb *blender*.

Capítol 6

Planificació

Inicialment el treball va ser planificat pensant en implementar una base que permetés generar models simples i a partir d'aquesta base anar ampliant el treball afegint funcionalitats. Finalment s'han afegit funcionalitats per importar models externs i exportar el model generat a format *.ply*.

6.1 Planificació inicial

La planificació inicial, que descriu com aconseguir una la base anteriorment descrita, esta plasmada en el diagrama de gantt de les Figures 6.1 i 6.2.

		Nombre	Duración	Inicio	Fin
0		**⊟Projecte**	**61d?**	**18/02/2013**	**13/05/2013**
1		Configuració de l'entorn de treball	1d?	18/02/2013	18/02/2013
2		Implementació interpret de gràmatiques	7d?	19/02/2013	27/02/2013
3		Test interpret	2d?	28/02/2013	01/03/2013
4		Documentació interpret	1d?	04/03/2013	04/03/2013
5		Implementació generador de models(basic)	15d?	05/03/2013	25/03/2013
6		Test generador	2d?	26/03/2013	27/03/2013
7		Documentacció generador	1d?	28/03/2013	28/03/2013
8		Implementació visualitzador	8d?	29/03/2013	09/04/2013
9		Test visualitzador	2d?	10/04/2013	11/04/2013
10		Documentació visualitzador	1d?	12/04/2013	12/04/2013
11		Composició del sistema generador+visualitzador	5d?	15/04/2013	19/04/2013
12		Test del sistema	4d?	22/04/2013	25/04/2013
13		Documentació del sistema	2d?	26/04/2013	29/04/2013
14		Generació conjunts de regles adequades per a la generació de ciu	6d?	29/04/2013	06/05/2013
15		Documantació regles	2d?	07/05/2013	08/05/2013
16		Redactat final de la memoria	3d?	09/05/2013	13/05/2013

Figura 6.1: Diagrama de gantt on apareixen les tasques necessàries per complir amb la planificació inicial.

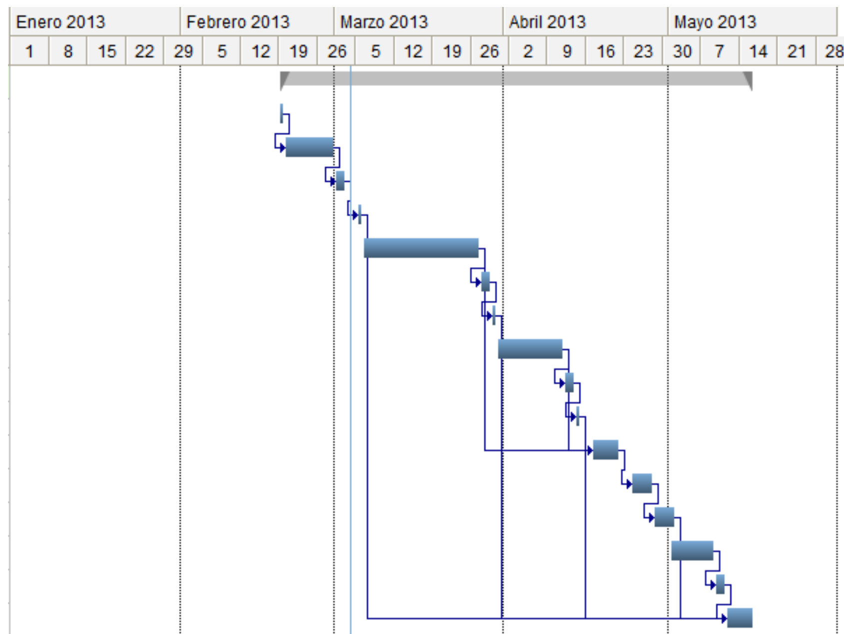

Figura 6.2: Diagrama de gantt on apareix l'ordre i el temps planificat per realitzar les tasques de la Figura 6.1.

6.2 Planificació real

L'obtenció d'aquesta base sobre la qual es volien afegir funcionalitats, no ha estat aconseguida en el temps estimat en la planificació original, degut a que la planificació final s'ha endarrerit respecte a la planificació inicial.

Al començament de la implementació van sorgir problemes derivats de voler usar *antlr* i *C++* , degut a la falta de documentació i funcionalitats que *antlr* proporciona sobre l'us d'antlr i *C++* . Aquest és un dels problemes que és va considerar al planificar el projecte, així tal com i és va proposar al realitzar l'anàlisi de possibles problemes, s'ha realitzat la implementació del parser i del generador en *Java*.

Els processos de test i implementació, és van subestimar ja que van sorgir errors inesperats, comportat un endarreriment afegit a l'anteriorment esmentat.Per tal de seguir amb la planificació inicial i poder acabar un generador bàsic al qual poder afegir funcionalitat,es va optar per no mantindre la planificació inicial respecte al proces de crear la documentació, en comptes d'anar generant la documentació mentre s'implementava el software, s'ha anat registrant tot allò que s'ha fet i provat per poder generar la documentació al final.

Tot i els retards esmentats, s'han pogut fer ampliacions al projecte com: permetre a l'usuari llegir i usar models auxiliars durant la generació visualització del model generat, i permetre exportar el model generat en format *.ply*, i aixi permetre usar els models generats en altres projectes.

6.3 Metodologia

La metodologia seguida per realitzar aquest projecte, ha sigut una metodologia iterativa, és va dissenyar una estructura simple, primer desenvolupant l'aplicació *ParseGen* que reconegues poques regles, i un visualitzador simple. A partir d'aquest punt de partida s'han anat afegint funcionalitats al visualitzador i a l'aplicació *ParseGen*, primer per aconseguir un software bàsic que complís els requisits mínim de ser capaç de generar models a partir d'una gramàtica, i finalment s'han anat a afegint al resta de funcionalitats.

Durant el proces de desenvolupament, a mesura que s'han anat afegint funcionalitats, s'ha anat validant e seu funcionament i resolent els problemes que presentaven. Per validar el sistema s'han anat creat petites gramàtiques simples d'exemple, que provessin les funcionalitats que s'afegien, i a mesura que

els sistema ha esdevingut més complex, s'han pogut provar gramàtiques mes complexes amb les quals finalment s'han pogut generar models que emulen l'estructura dels edificis.

Capítol 7

Anàlisi de Costos

S'analitzaran els costos que comportarà la realització del projecte, aquests costos seran calculats com si una empesa hagués de dur a terme el projecte, ja que realment el cost d'aquest projecte serà quasi nul al ser desenvolupat per un estudiant, en tot cas és podria comptar la despesa en electricitat. Les quantitats monetaries que s'exposen són orientatives.

7.1 Costos de hardware

Son els costos corresponents a l'us(amortització) de l'equip informàtic que s'usarà per la realització d'aquest projecte, s'assumeix que és necessita un total de 3 anys per amortitzar l'equip informàtic.

S'assumeix que s'usarà un ordinador de gama alta de 1200€, per tant el cost d'amortització per mes serà de 33€.

Concepte	cost mensual	Temps	Import
Equip informàtic	33€	3 mesos	€100

Figura 7.1: Taula que mostra els costos hardware del projecte

7.2 Costos software

El cost total del software usat és de 0€, ja que tots els paquet usats són amb llicencia open source i el sistema operatiu sobre el que es desenvolupara també ho és, per tant no hi ha cap cost per la part del software emprat.

7.3 Costos de personal

Aquests son els costos de la ma d'obra que s'ha d'utilitzar per a completar el projecte. Per tal de simular el fet que el projecte es dut a terme per una empresa de software és tindran en compte els rols habituals en un projecte d'aquestes característiques. He suposat que és una empresa la que vol realitzar el projecte per si mateixa i no per un client concret, aixi que s'elimina la necessitat d'interacció del client.

La estimació dels costos de personal dependrà en gran mesura de l'estimació temporal que s'ha fet prèviament per a cada tasca a realitzar,d'acord amb la planificació i el diagrama de gantt descrit a la secció de planificació temporal.En aquest cas concret prenem que un dia compren una jornada laboral de 8 hores.

Rol	Hores	Preu/hora	Total
Director del projecte	57	50€	2850€
Dissenyadors	182	25€	4550€
Programadors	210	20€	4200€
Testers	40	12€	480€
Preu Total			12080€

Figura 7.2: Taula que mostra els costos de Personal

Descripció dels rols:

- Director del projecte: és l'encarregat de coordinar i organitzar l'equip de treball, validar la correcta realització del projecte i reunir-se amb els dissenyadors per establir les funcionalitats que ha de complir el producte resultant.
- Dissenyadors: són els encarregats de especificar i dissenyar l'aplicació perquè posteriorment sigui implementada pels programadors, i també generar la documentació tècnica del projecte.

- Programadors: són els encarregats d'implementar l'aplicació i de la posterior correcció d'errors que es puguin trobar.
- Testers: son els encarregats de validar el correcte funcionament de l'aplicació(testing).

7.4 Costos de despeses generals

Els Costos de despeses generals corresponen als costos degut a l'ús d'equipaments i serveis com la factura de la llum, Internet, etc. Si suposem que una empresa és la que realitza el projecte, és suposarà que te una oficina llogada en un business center. L'avantatge de fer aquest suposició és que en el preu del lloguer ja venen inclosos els serveis com la llum i Internet.

Habitualment una empresa te mes d'un projecte i el cost del lloguer es reparteix entre els projectes que s'hi duen a terme, encara que per fer els nostres càlculs suposarem que no tenim altres projectes en desenvolupament.

Com que el lloguer s'ha de pagar per a cada mes que s'usen les instal·lacions sent el mes la fracció de pagament, aquí haurem de posar un total de quatre mesos, ja que el projecte començarà a mitjans de febrer i acabarà a mitjans d'abril.

Concepte	cost mensual	Temps	Import
Lloguer	1200€	4 mesos	4800€

Figura 7.3: Taula que mostra els costos de despeses Generals

7.5 Costos totals

Finalment calculem el cost total del projecte sumant els costos calculats anteriorment.

Concepte	Import
Costos de Personal	12080€
Costos de Hardware	100€
Costos de Software	0€
Costos de despesa general	4800€
Cost Total	16980€

Figura 7.4: Taula que mostra el cost total del projecte

Es podria sumar l'increment de l'IVA, però com ja hem dit el projecte no està destinat a cap client i per tant no és un producte que és comercialitzi.

7.6 Anàlisi del cost mediambiental

Un projecte software en general te un cost mediambiental acumulat derivat dels recursos consumits durant el desenvolupament: energia elèctrica, equips informàtics i els costos mediambientals derivats de la producció d'aquesta energia i dels equips. Peró podem afirmar que en comparació amb altres projectes d'àmbit industrial és un cost negligible.

A més a més les eines com la desenvolupada en aquest projecte, permeten reduir el temps dedicat a la generació d'escenaris tridimensionals i aixi reduir costos de desenvolupament en altres projectes. Es pot afirmar que l'us d'eines de generació procedural permeten reduir el cost econòmic i mediambiental associat als projectes que les usen.

Capítol 8

Conclusions

Al ser un problema del mon ”real” i no un exercici delimitat ha permès a l'estudiant enfrontar-se amb dificultats típiques al desenvolupar una aplicació des de zero. Com per exemple, la tria del software que s'usarà i posteriorment la obtenció de la informació, necessària per usar aquest software, que en molts casos és escassa o nul·la.

Tot i ser un projecte orientat a la llarga a reduir costos i a facilitar l'etapa de creació de contingut, després de treballar amb l'eina i la gramàtica, s'ha fet evident que es necessari posseir un ventall de coneixements més amplis que els d'un artista, per poder generar models mitjanament complexos, a més a més dels coneixements propis d'un artista és necessiten coneixements del funcionament d'una gramàtica i com amb aquesta poder generar models. Per tant qualsevol empresa o grup que vulgui utilitzar l'eina per generar continguts haurà d'entrenar els seus artistes, perquè siguin capaços de crear gramàtiques suficientment bones per generar edificis.

Caldria destacar que tot i ser un sistema enfocat a la creació d'edificis, com el propi nom del projecte indica, usant la gramàtica es possible crear altres tipus d'elements que no tenen perquè ser edificis.

8.1 Treball futur

Aquest és un projecte que es pot ampliar molt i de diverses maneres, tan per la part dels models generats, com per la posterior visualització dels mateixos al crear models més realistes.

Per tal d'incrementar el detall se'ls hi haurien d'afegir textures, i posteriorment per tal que les textures no és trepitgin entre si degut a la intersecció entre els diversos models que formen la figura, afegir concepció de les interseccions entre els sub-models que constitueixen el model generat.

Actualment, els models generats i exportats contenen geometria oculta, fruit d'exportar tots els vertex que el conformem, independentment de si estan ocults per altres polígons que conformen el model. Es podria optimitzar la quantitat de geometria a exportar aplicant un filtratge al model generat per eliminar tota aquesta geometria oculta, també seria interessant afegir tests d'oclusió durant el proces de generació del model, per exemple poder decidir si una part de paret és visible o no des de l'exterior per afegir o no una finestra al model.

De cara a afegir complexitat a les escenes i a millorar el visualitzador, es podria oferir a l'usuari dos modes de visualització, un com l'actual per visualitzar el model amb detall, i un altre per construir una escena a partir dels models que es generen, per poder observar els models en conjunt, fins i tot construir permetre ciutats, combinant models fruit de l'aplicació de gramàtiques diferents. També seria recomanable l'us de tècniques avançades d'il·luminació per poder visualitzar escenes més realistes i també millorar el rendiment del visualitzador.

I finalment per afavorir la interacció amb l'usuari, seria positiu oferir un entorn on l'usuari pogués dissenyar les gramàtiques, dins del mateix visualitzador, i que mostres a l'usuari els errors comesos en la creació de la gramàtica igual que un IDE modern.

Bibliografia

[1] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 614–623, New York, NY, USA, 2006. ACM.

[2] Paul Bourke. Ply - polygon file format. `http://paulbourke.net/dataformats/ply`.

[3] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann Publishers Inc, San Francisco, CA, USA, third edition, 2002.

[4] A. Lagae, S. Lefebvre, R. Cook, T. DeRose, G. Drettakis, D. S. Ebert, J. P. Lewis, K. Perlin, and M. Zwicker. State of the art in procedural noise functions. In *EG 2010 - State of the Art Reports*, 2010. to appear.

[5] Ken Perlin. Improving noise. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 681–682, New York, NY, USA, 2002. ACM.

[6] Adam Runions, Brendan Lane, and Przemyslaw Prusinkiewicz. Modeling Trees with a Space Colonization Algorithm. In D. Ebert and S. Mérillou, editors, *Eurographics Workshop on Natural Phenomena*, 2007.

[7] Stefan Greuter, Jeremy Parker, Nigel Stewart, and Geoff Leach. Real-time procedural generation of 'pseudo infinite' cities. In *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, GRAPHITE '03, pages 87–ff, New York, NY, USA, 2003. ACM.

[8] Tao Jiang, Ming Li, Bala Ravikumar, and Kenneth W. Regan. Algorithms and theory of computation handbook. chapter Formal grammars and languages, pages 20–20. Chapman & Hall/CRC, 2010.

[9] George Stiny, James Gips, George Stiny, and James Gips. Shape grammars and the generative specification of painting and sculpture. In *Segmentation of Buildings for 3DGeneralisation. In: Proceedings of the Workshop on generalisation and multiple representation , Leicester*, 1971.

[10] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.