EPS
Escola Politècnica Superior
UdG

Projecte/Treball Fi de Carrera

Estudi: Eng. Tècn. Informàtica de Sistemes. Pla 2001

Títol: Modelat d'edificis basats en extrusions procedurals per a skylineEngine

Document: Memòria

Alumne: Marc Pont Casadevall

Director/Tutor: Gustavo Patow
Departament: Informàtica i Matemàtica Aplicada
Àrea: LSI

Convocatòria (mes/any): Setembre 2012

Índex de continguts

1. Introducció

Actualment ens trobem en un món on tot gira al voltant de les noves tecnologies, i un pilar fonamental és l'oci i l'entreteniment. Això engloba principalment les indústries del cinema, videojocs i realitat virtual.

Un dels problemes que tenen aquestes indústries és com crear l'escenari on es produeix la història. En un videojoc, per exemple, es necessita crear tot el món virtual. Normalment serà una o més ciutats on es troba el personatge. Al principi l'única opció que hi havia era muntar la ciutat de manera manual, o sigui dissenyar cada edifici, un per un, amb la conseqüència d'una despesa important de temps i diners.

Per a solucionar aquest problema cada vegada s'utilitzen més eines informàtiques. L'avantatge de l'eina informàtica es deixar que l'ordinador s'encarregui de realitzar les tasques difícils i l'usuari només ha de modificar els paràmetres per obtenir el resultat desitjat.

El modelatge procedural és una eina informàtica que, tal com s'ha explicat anteriorment, soluciona el problema de generar ciutats virtuals. L'ordinador s'encarrega de generar la ciutat automàticament, donats els paràmetres que hagi introduït l'usuari.

Cada eina destinada al modelatge procedural té les seves particularitats. Per posar un exemple, la eina desenvolupada en aquest projecte disposaria d'una interfície per introduir els paràmetres de com es vol crear l'edifici, com per exemple, número de finestres, balcons, forma de la teulada, etc. L'usuari escolliria tots els paràmetres, i l'ordinador s'encarregaria d'obtenir el resultat.

D'aquesta manera s'aconsegueix molta rapidesa i eficiència al crear la ciutat, i no es necessita un expert que vagi dissenyant cada edifici, un per un.

Figura 1.1 Exemple de modelatge procedural.

1.1. Motivacions personals

Dins la informàtica sempre m'ha interessat la branca dels gràfics, tant videojocs, realitat virtual o cinema. Només he tingut l'oportunitat de fer 2 assignatures orientades a aquesta branca. Degut a aquest interès he volgut involucrar-me a desenvolupar un projecte d'aquest estil, per poder aprendre i adquirir experiència sobre aquest tema.

1.2. Propòsits i objectius

L'objectiu d'aquest projecte de final de carrera és crear una eina integrada al **skyline**Engine, que serveixi per crear edificis de manera procedural, on l'usuari pugui definir l'estètica d'aquest edifici, introduint la seva planta i els perfils adequats.

El que s'implementarà serà una eina de modelatge per a dissenyadors, que a partir d'una planta i perfils pugui crear l'edifici.

Aquest projecte es desenvoluparà a sobre del mòdul de generació d'edificis del **skyline**Engine, una eina pel modelatge de ciutats que s'executa sobre el Houdini 3D, que és una plataforma genèrica pel modelatge procedural d'objectes.

El desenvolupament d'aquest projecte implica:

- Estudi de la plataforma de desenvolupament Houdini 3D i de les llibreries necessàries per la incorporació de scripts Python. Estudi de les EEDD internes de Houdini.
- Aprendre i manejar el llenguatge de programació Python.
- Estudi del codi de l'article Interactive Architectural Modeling with Procedural Extrusions, per en Tom Kelly i en Peter Wonka, publicat a la revista ACM Transactions on Graphics (2011).
- Desenvolupament d'algorismes de conversió de geometria d'una estructura tipus face-vertex a una de tipus half-edge, i viceversa.
- Modificació del codi Java per acceptar crides sense interfície d'usuari i amb estructures de dades generades des de Python.
- Aprendre el funcionament de la llibreria JPype per permetre enllaçar el Java dins el Python.
- Estudi del skylineEngine i de les llibreries per la creació d'edificis.
- Integració del resultat dintre del skylineEngine.
- Verificació i ajust de les regles i paràmetres de la simulació per a diferents edificis.
- Arrodoniment de la documentació del treball realitzat.

1.3. Estructuració del document

Aquest document s'ha organitzat en 15 capítols, que recullen tota la documentació i explicació necessària per la comprensió del projecte.

1. Introducció, motivacions, propòsits i objectius del projecte. En aquest capítol s'introduirà el tema del projecte, el perquè del desenvolupament, quins són els objectius i com s'ha organitzat tot el procés de desenvolupament.
2. Estudi de la viabilitat. En aquest capítol es justifiquen els paràmetres que fan possible el desenvolupament del projecte.
3. Metodologia. Explicació de la metodologia utilitzada.
4. Planificació. Es definirà l'estratègia seguida per arribar als objectius plantejats.
5. Marc de treball i conceptes previs. En aquest capítol es descriuen els diversos aspectes relacionats amb el desenvolupament general del projecte, que serviran per entendre millor els següents capítols. També es tractaran les principals accions desenvolupades durant les primeres etapes del projecte. S'inclouran els passos d'estudi i aprenentatge de conceptes que s'hagin utilitzat pel desenvolupament.
6. Requisits del sistema. Es definiran els requisits del software, de manera que es vegin quins són els objectius de l'aplicació juntament amb les seves funcionalitats que es volen aconseguir. Aquest capítol permetrà entendre els elements que rodegen el sistema informàtic que s'intenta construir.
7. Estudis i decisions. Es descriuran les eines utilitzades, les seves característiques i l'ús que se'ls hi ha donat.
8. Anàlisi i disseny del sistema. Aquest capítol proporciona una comprensió precisa de les necessitats del sistema, és a dir, s'encarrega de la investigació del problema a resoldre, però no s'interessa en trobar una solució. Usant l'enginyeria del software, es traduiran els requisits nombrats en capítols anteriors a un llenguatge més formal. La part de disseny permet augmentar augmentar el nivell d'especificació, i realitzar un esquema d'implementació del sistema mitjançant diverses eines de programació orientada a objectes.
9. Implementació i proves. En aquest capítol es donen a conèixer com s'ha implementat l'aplicació, les classes i els mètodes que resulten més significatius per a la comprensió de funcionament de l'aplicació.
10. Implantació i resultats. En aquest capítol es mostres proves d'execució de l'aplicació, es mostren imatges dels edificis resultants i tot allò que es pugui visualitzar del conjunt que ha estat implementat.
11. Conclusions. En aquest capítol s'exposaran les conclusions obtingudes un cop finalitzat el treball.
12. Treball futur. En aquest capítol s'exposa tot el que es podria fer per millorar l'aplicació o ampliar-la de forma interessant.
13. Bibliografia. Aquest capítol conté les referències utilitzades pel desenvolupament del projecte.

14. Annexos. Aquest capítol conté les definicions dels tecnicismes més freqüentment utilitzats, així com explicacions i experiments no indispensables per a la comprensió global del projecte.
15. Manual d'usuari i/o instal·lació. Aquest capítol conté les especificacions del funcionament del producte.

2. Estudi de la viabilitat

Per desenvolupar el projecte no es requereix d'una gran infraestructura i els costos d'estructura són limitats.

Els materials amb els quals es contava des d'un principi són els següents:

- Ordinador amb sistema operatiu Windows XP 32 bits.
- Houdini 3D de Sidefx, amb llicència d'aprenentatge gratuïta.

La valoració dels costos econòmics del projecte es separa en dues parts, els costos de recursos humans i els costos per maquinària.

2.1. Recursos humans

En aquest projecte ha estat necessari la participació de 2 perfils, el d'analista pel tema del disseny, i el programador per la part d'implementació.

Per calcular els costs dels recursos humans s'ha comptat de la següent manera:

- Cost analista: 15€ / Hora
- Cost programador 10€ / Hora

TASCA	PERFIL	HORES	COST
Estudi Houdini	Analista	20	300
Estudi Python	Analista	15	225
Estudi llibreria procedural extrusions	Analista	30	450
Estudi enllaçar Python-Java-ProcExtr	Analista	30	450
Implementació Wrapper Java	Programador	40	400
Implementació en codi Python	Programador	30	300
Integració al Houdini	Programador	15	150
Proves i millora del codi	Programador	10	100
Documentació	Analista	30	450
TOTAL		**220**	**2825**

2.2. Maquinària

S'ha utilitzat tant un ordinador sobretaula com un portàtil per anar desenvolupant el projecte, s'ha considerat que el preu del sobretaula és de 1000€ i el portàtil de 450€ o sigui que el cost total per maquinària ha estat de 1450€.

2.3. Cost total

No hi ha hagut costos per llicències ja que s'han utilitzat versions llicenciades per a estudiants, com per exemple la versió apprentice de Houdini, o bé s'ha utilitzat freeware, així doncs el cost total del projecte ha estat de 4275€.

3. Metodologia

En aquest projecte s'ha seguit una metodologia específica del projecte skylineEngine:

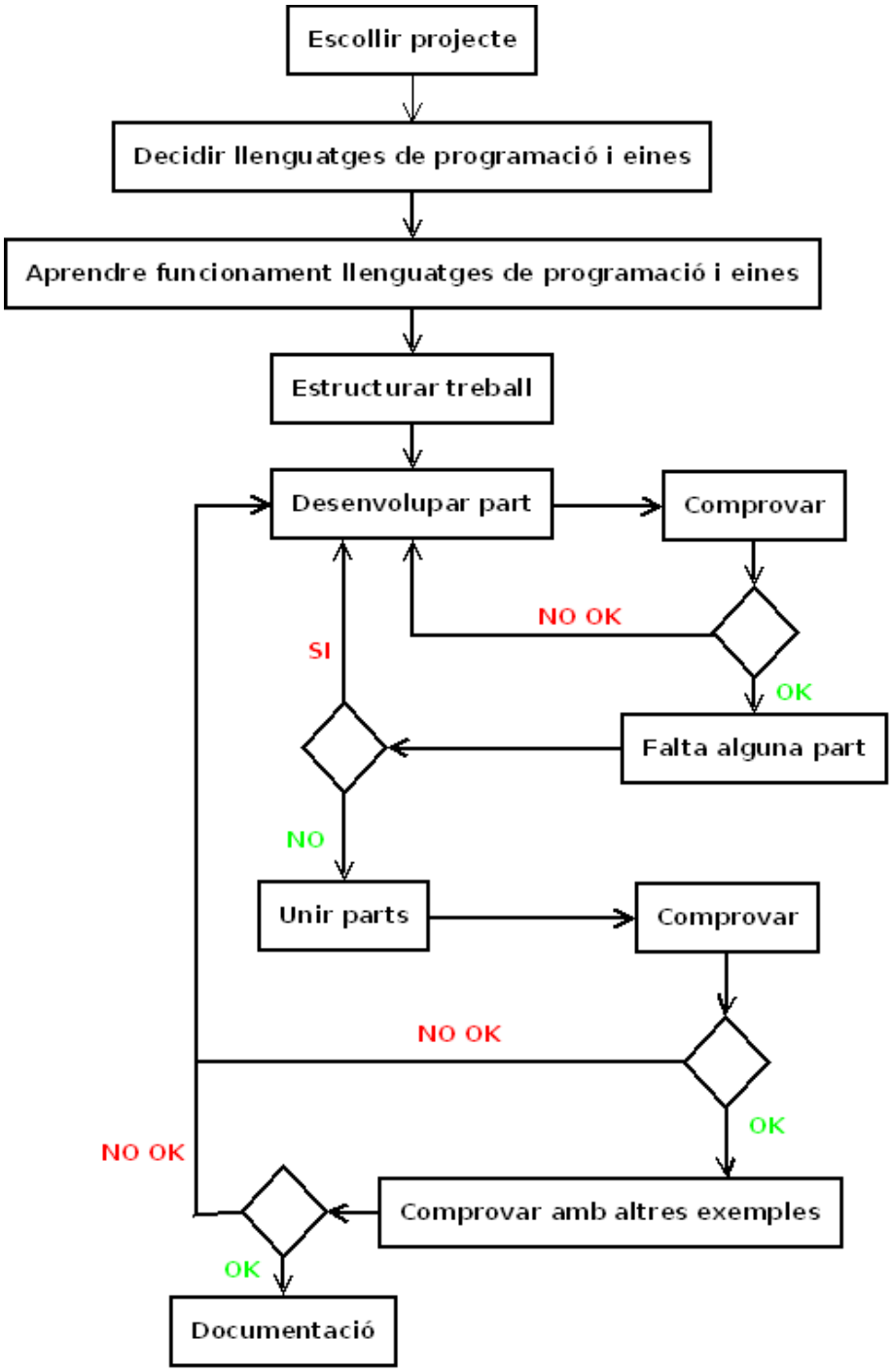


Figura 3.1 Diagrama de flux de la metodologia skylineEngine.

1. Escollir el projecte a desenvolupar.
2. Decidir el llenguatge de programació i les eines a utilitzar.
3. Aprendre el llenguatge de programació i el funcionament de les eines escollides.
4. Estructurar el treball en parts segons les funcions que ha de realitzar.
5. Desenvolupar la part corresponent seguint l'ordre de l'estructura del treball.
6. Fer comprovacions per tal de confirmar que el funcionament és el correcte al finalitzar la part.

    6.1. Si al fer les comprovacions el resultat no és l'esperat, es torna al punt 5 per a realitzar els canvis oportuns en la última part desenvolupada o en les anteriors, si és necessari.

    6.2. Si al fer les comprovacions el resultat és l'esperat, es desenvolupa la part següent tornant al punt 5, en cas que s'hagin finalitzat les parts amb les seves respectives comprovacions s'inicia el punt 7.

7. Unir totes les parts desenvolupades i comprovar que el funcionament és correcte.

    7.1. Si al fer les comprovacions el resultat no és l'esperat, es torna al punt 5 per a realitzar els canvis oportuns en l'última part desenvolupada o en les anteriors, si és necessari.

    7.2. Si al fer les comprovacions el resultat és l'esperat, s'inicia el punt 8.

8. Generar diferents models d'exemple per a comprovar que el funcionament és el correcte.

    8.1. Si al fer les comprovacions el resultat no és l'esperat, es torna al punt 5 per a realitzar els canvis oportuns en l'última part desenvolupada o en les anteriors, si és necessari.

    8.2. Si al fer les comprovacions el resultat és l'esperat s'inicia el punt 9.

9. Acabar la documentació.

D'aquesta manera es veu com el que s'intenta és dividir el projecte en mòduls per poder organitzar el temps de desenvolupament, verificació i correcció de cada part.

Durant el llarg del desenvolupament del projecte es fa un seguiment mitjançant tutories setmanals o bisetmanals depenent de l'etapa, ja que en l'inici la manera d'avançar sol ser més lenta que al final, i per tant no era necessari fer tutories setmanalment.

Sempre que s'acabi un mòdul s'ha d'intentar tancar-lo si es pot, i no modificar-lo més per tal de garantir que funciona correctament sense errors, i així en etapes futures podrem assegurar que els mòduls anteriors estan funcionant correctament.

La tècnica que s'ha seguit per a la creació d'aquesta aplicació és la de disseny descendent. Per això hem utilitzat com a metodologia el llenguatge UML (Llenguatge de

Modelatge Unificat o Unified Modeling Language), que tot i no ser una metodologia, és un llenguatge de modelat estàndard pel camp de l'enginyeria del programari. L'UML s'utilitza per definir un sistema, per detallar els seus elements, per documentar i construir. Per aconseguir això, l'UML disposa de nombrosos tipus de diagrames que mostren diversos aspectes dels elements representats.

4. Planificació

En tot projecte, és molt important planificar bé les parts del projecte per poder fer una valoració del temps que s'emprendrà pel seu desenvolupament. En aquest capítol es descriurà la planificació inicial que es va intentar seguir, però per alguns problemes tècnics es va haver de modificar lleugerament.

4.1. Planificació inicial

Aquest projecte es va començar cap al Novembre del 2011 i inicialment s'havia plantejat acabar-lo per el Juliol del 2012, però finalment s'ha allargat fins al Setembre de 2012.

Els 2 primers mesos serien dedicats a la investigació i aprenentatge del funcionament de la plataforma Houdini i a la integració del Python amb el Houdini a través dels SOP (Surface Operator de houdini).

Cap a finals de Gener es començaria a pensar una manera d'enllaçar el Python amb el Java, per tant aprendre com fer-lo funcionar. Paral·lelament a aquesta etapa, es començaria a investigar com funcionava el codi de la llibreria d'extrusions procedurals, per descobrir com es podria reutilitzar pels nostres propòsits.

Al Març, un cop trobats els punts d'entrada del codi, es procediria a fer la implementació en Java d'una classe “Wrapper” que seria cridada des de Python per que utilitzés la llibreria d'extrusions procedurals. Per tant el “Wrapper“ rebria els paràmetres des de Houdini (planta i perfils), per cridar la llibreria d'extrusions procedurals i rebre els resultats per tornar-ho a enviar a Houdini.

Cap a mitjans de Maig, quan s'hagués completat la implementació amb Java es faria la implementació amb Python per cridar la classe en Java realitzada, i tot seguit s'aniria comprovant que funcionessin correctament les connexions entre Python la classe Wrapper i la llibreria d'extrusions procedurals. Una vegada acabat això, començarem amb la implementació dins el Houdini.

Finalment s'acabarien de corregir els errors tècnics, millora del codi i finalitzar la memòria del treball.

El pla de treball es pot definir amb 7 tasques:

- Planificació inicial del projecte. Com a primera tasca realitzar una planificació inicial per decidir què fer i com fer-ho.
- Aprenentatge. La següent tasca seria instal·lar les eines necessàries i aprendre com funcionen, en concret el Houdini, Python i l'enllaç entre Python i Java.
- Aprenentatge d'un codi original. Investigar la llibreria d'extrusions procedurals, per aprendre a utilitzar el seu codi per als nostres objectius.
- Disseny i implementació en Java per enllaçar amb el codi original el que faci falta.
- Disseny i implementació amb Python per enllaçar-ho amb Java de manera que des

de Python es puguin enviar paràmetres cap a la implementació en Java i al final recollir els resultats que s'han obtingut en Java.

- Integració al Houdini.
- Documentació. Escriure la memòria del treball.

4.2. Temps estimat

Es va planejar que el projecte tingués una durada d'uns 8 mesos. A continuació es mostra el digrama de gantt:

Figura 4.1 Diagrama de Gantt de la planificacií inicial

5. Marc de treball i conceptes previs

En aquest capítol s'explicaran els detalls necessaris per tal de poder entendre els algorismes implementats i les eines utilitzades en el projecte.

5.1. Modelatge procedural

Es denomina modelatge procedural a la tècnica per a crear models 3D a partir d'un conjunt de regles en comptes de construir a partir de primitives geomètriques, com cubs o esferes per exemple.

Aquest conjunt de regles, o bé pot estar incorporat en el codi conFigurable per paràmetres, o bé pot ser independent del motor d'avaluació. La sortida s'anomena contingut procedural, que pot ser usat en videojocs, pel·lícules o per l'usuari si vol editar el contingut de forma manual.

Totes les tècniques de modelatge per computador requereixen algorismes per gestionar i emmagatzemar dades en algun moment.

Per tant, el modelatge procedural se centra en la generació d'un model basat en un conjunt de regles, en lloc de l'edició del model a través de l'entrada de l'usuari. Com a conseqüència, el modelatge procedural sovint s'aplica quan la feina de crear un model 3D utilitzant modeladors és massa feixuga, o quan es requereixen eines més especialitzades. Aquest cas sol ser el de les plantes, arquitectura o paisatges.

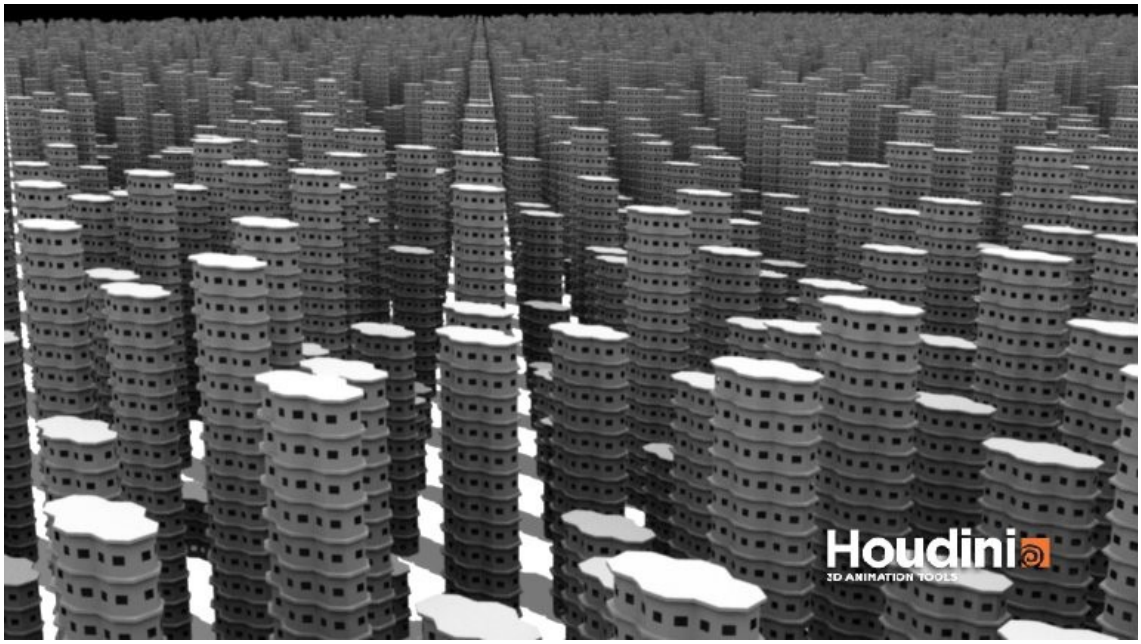


Figura 5.1 Exemple d'una ciutat generada per modelatge procedural

5.2. SkylineEngine

SkylineEngine és una eina de modelatge procedural urbà desenvolupat com un banc de proves per a nous algoritmes i tècniques de modelatge. Aquest sistema presenta algunes característiques noves que el converteixen en un sistema únic, com un paradigma basat en grafs que permet a l'usuari crear tant contingut ric en ciutats amb districtes diferents, carreteres principals i secundàries, els blocs, lots i edificis, com també altres elements urbans com carrers, voreres, parcs, ponts i monuments.

skylineEngine és un sistema desenvolupat al Grup de Geometria i Gràfics de la Universitat de Girona, que està basat de diferents mòduls. Tot en **skyline**Engine està fet per mòduls, alguns d'ells en codi Python, altres en forma de Houdini Digital Assets (HDA's). Dintre del **skyline**Engine, el modul buildingEngine ha estat concebut com a una eina de renderització i modelatge procedural d'edificis i està escrit en una barreja de Python i Houdini Digital Assets.

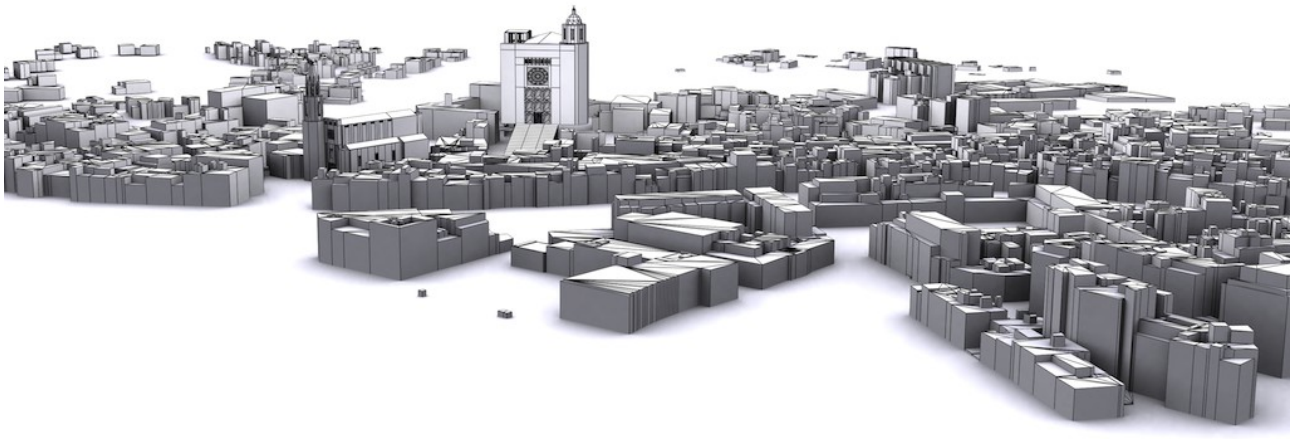


Figura 5.2 La ciutat de Girona generada per SkylineEngine

5.3. Extrusions procedurals

La extrusió és un procés utilitzat per crear objectes amb secció transversal definida i fixa. A partir d'aquesta secció transversal l'objecte es pot estirar o arronsar, donant a l'objecte el volum desitjat.

Figura 5.3 Exemple d'extrusió en la realitat

En la Figura 5.3 es pot veure un dels exemples que hi ha a la realitat, com es veu, té una secció transversal fixa, amb forma de X, i l'objecte ser més llarg o més curt, però manté la mateixa secció.

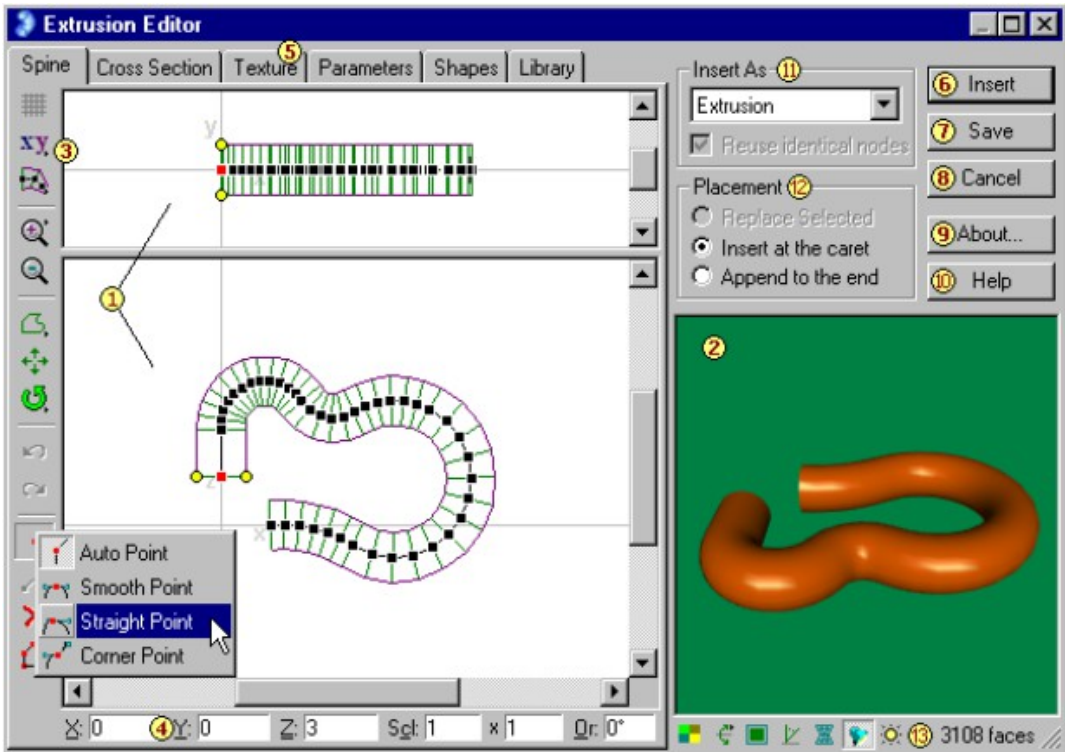


Figura 5.4 Programa que genera objectes utilitzant l'extrusió

En la Figura 5.4 podem veure un altre exemple on es mostra un programa que pot realitzar la tècnica de fer extrusions. La secció també es manté fixa però incorpora una nova funcionalitat que permet modificar la orientació d'aquesta secció, D'aquesta manera es pot formar un tub curvilini.

5.3.1. Extrusions clàssiques

La tècnica d'extrusions fa molt de temps que s'utilitza, però té la restricció que la secció sempre és fixa. Això implica que utilitzant extrusions no es podrà crear qualsevol objecte i en el moment que es vulgui editar la secció, no es podrà fer.

Com es veu en els exemples de les figures 5.3 i 5.4, la secció no canvia. En tot cas en el segon exemple s'ha pogut canviar la orientació de la secció.

Una altra funcionalitat que permet l'extrusió clàssica és ampliar o reduir la secció, però sempre de manera regular, o sigui una secció quadrada de 1m cada costat, es podria reduir a 50cm cada costat, per exemple. Això permetria aprimar o engrandir un tub, però sempre de forma regular. Per tant aquesta tècnica no resulta gaire útil per generar objectes complicats. Per exemple un edifici, si és molt irregular, serà impossible de crear.

Primer de tot es veurà un exemple on si que es podrà generar un edifici utilitzant extrusions clàssiques en un edifici geomètricament senzill. Veure Figura 5.5.

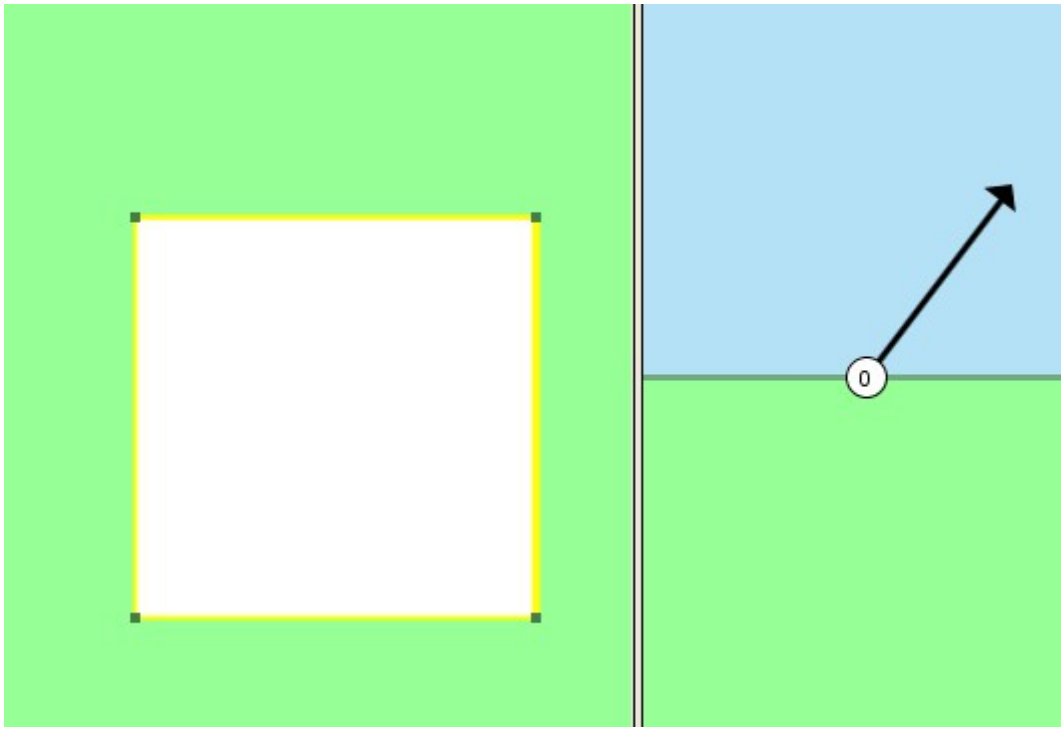


Figura 5.5 Planta quadrada amb perfil inclinat

La planta és un quadrat, i el perfil és inclinat, per tant el resultat esperat és el de una piràmide. El fet important d'aquest exemple, és que, amb extrusions clàssiques, es pot aconseguir aquest resultat, ja que l'únic que s'ha de fer és anar reduint la secció de la planta a un punt al centre, formant la piràmide.

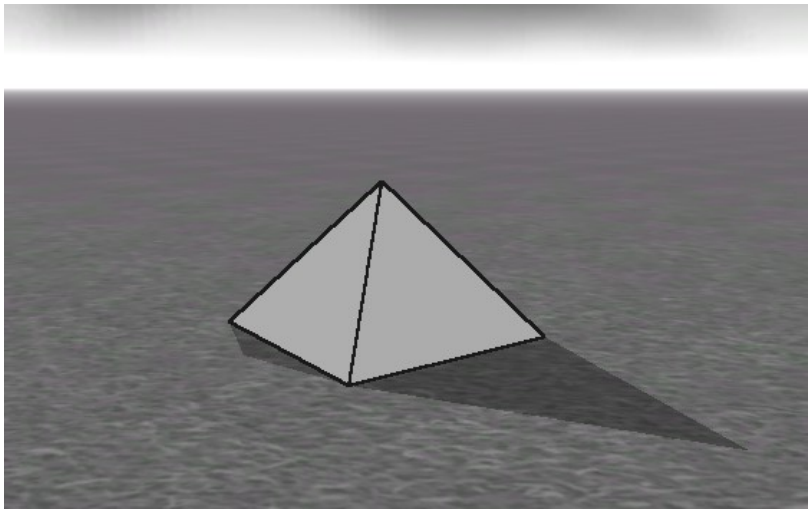


Figura 5.6 Resultat de l'extrusió anterior

Però a la Figura 5.7 veiem un exemple on aquesta tècnica no pot funcionar.

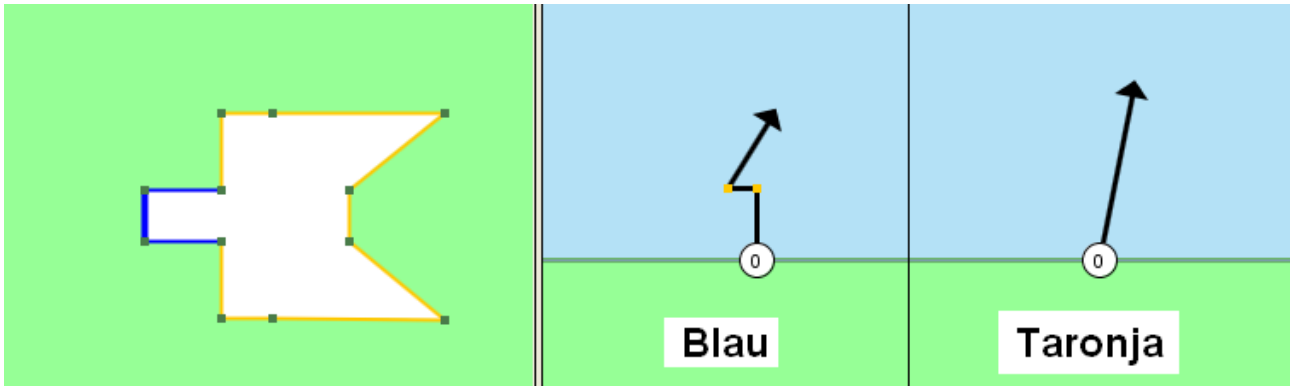


Figura 5.7 Exemple amb geometries més complexes

En aquest exemple, la planta ja té una forma complexa. Per tant, seria inviable utilitzar les extrusions clàssiques: no es pot simplement anar reduint la secció i anar pujant creant l'edifici, com es feia amb la piràmide.

A més s'han definit 2 tipus de perfils, el blau i el taronja corresponent a les respectives parts de la planta. A la Figura 5.8 es veu el resultat.

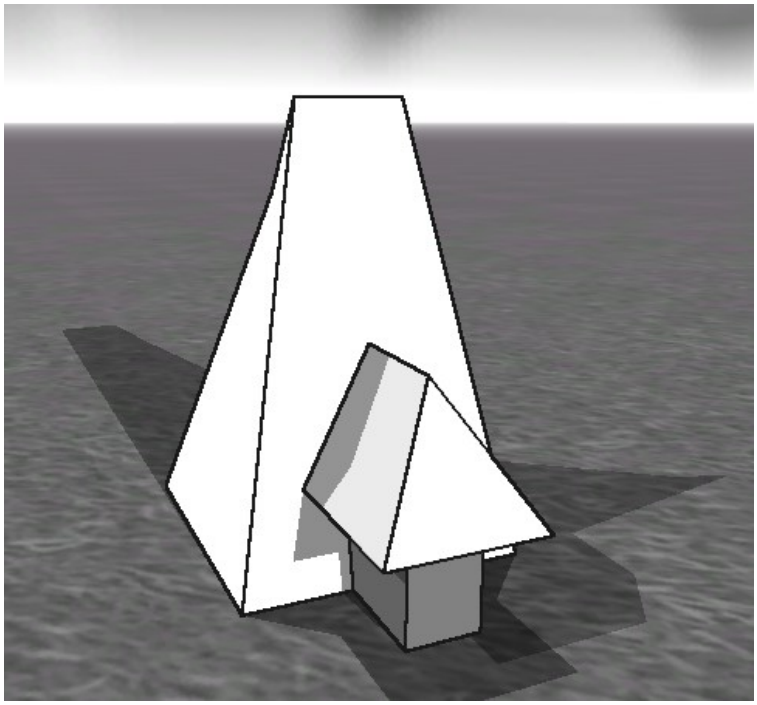


Figura 5.8 Resultat de l'extrusió de l'exemple anterior

Com s'ha dit anteriorment aquest resultat no es pot fer utilitzant les extrusions clàssiques, sinó utilitzant el mètode straight skeleton que s'explicarà tot seguit.

5.3.2. Straight Skeleton

Straight Skeleton (esquelet recte) és un mètode de representació d'un polígon per un esquelet topològic. És similar en alguns aspectes a l'eix medial, però difereix en què l'esquelet està compost de segments de línia recta, mentre que l'eix mitjà d'un polígon pot implicar corbes parabòliques.

L'Straight Skeleton d'un polígon està definit per un procés continu en el que la reducció de les vores del polígon es mouen cap a dins paral·lels a si mateixos a una velocitat constant.

Així doncs, els vèrtexs també es mouen a velocitats que depenen de l'angle del vèrtex. Si un d'aquests vèrtexs en moviment xoca amb una vora adjacent, el polígon es divideix en dos per la col·lisió, i el procés continua en cada part. L'esquelet recte és el conjunt de corbes traçades pels vèrtexs que es desplacen en aquest procés.

En la Figura 5.9 es pot veure com funciona aquest mètode:

- En la imatge i) es mostra el polígon resultant de fer el mètode de Straight Skeleton. Cada vèrtex exterior té una aresta que va en direcció cap al centre del polígon i com que té una forma irregular, es veu que no intersequen tots en un mateix punt al

mig, sinó que cada un va a parar a la intersecció que li toca.

- En la imatge ii) es mostra el procediment que s'ha seguit per donar el resultat de la imatge i) tal i com s'ha explicat en l'apartat anterior: cada vèrtex exterior es va apropant cap al centre del polígon fins que intersequi amb un altre vèrtex que també s'està apropant, o amb aresta.
- Finalment la imatge iii) mostra el resultat final representant volum a la figura després d'haver fet el Straight Skeleton. Recordar que aquest procés a més de consistir en apropar els vèrtexs i arestes cap a l'interior també creix cap amunt per donar lloc a una Figura amb volum.

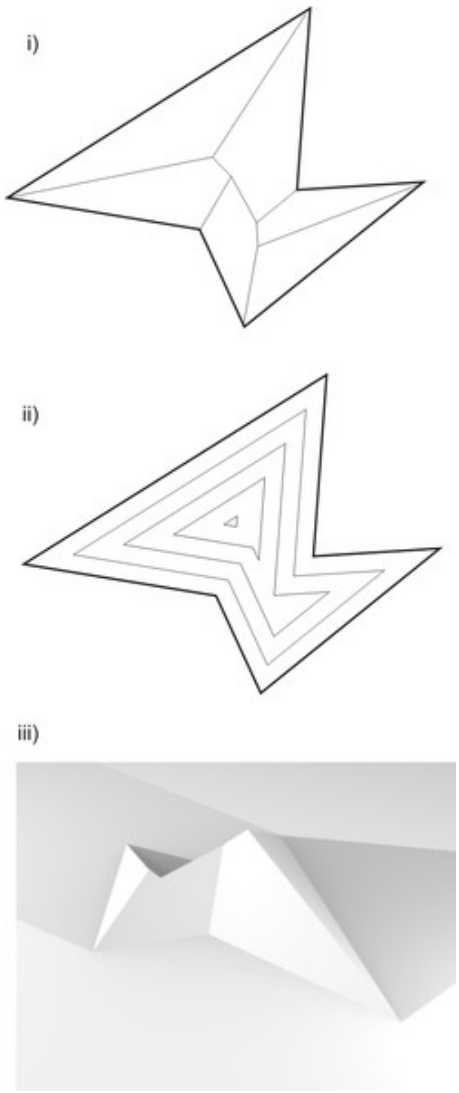


Figura 5.9 Exemple de Straight Skeleton

6. Requisits del sistema

Els requeriments del sistema es divideixen bàsicament en 2 tipus:

- Requeriments funcionals: Descriuen quins són els serveis que ofereix l'aplicació, independentment de la implementació.
- Requeriments no funcionals: Són aquells que fan referència a les restriccions del tipus de disponibilitat de recursos, seguretat, etc. Si complim els requeriments no funcionals es garanteix que es podrà executar l'aplicació sense problema.

6.1. Requisits funcionals

L'objectiu d'aquest projecte és generar el volum d'un edifici basat en el mètode d'extrusions a partir de 2 tipus de paràmetres: Una planta, i un o més perfils. A més, un cop s'hagi creat l'edifici, s'integrarà amb l'skylineEngine de manera que es puguin afegir les funcionalitats corresponents a l'edifici resultant.

L'usuari tindrà disponible una interfície d'usuari per tal de poder crear o editar geometries de plantes o perfils, de manera que es podrà editar l'edifici resultant en temps real modificant els paràmetres d'entrada, planta o perfils, aquesta interfície serà la que proporciona el Houdini.

Els paràmetres tindran les següents precondicions:

- La planta haurà d'estar definida dins el pla XZ. Si l'usuari incorpora una planta que es defineixi dins les 3 dimensions, el programa eliminarà la component Y, convertint la planta introduïda en una que només estigui definida dins el pla XZ. Veure Figura 6.1.

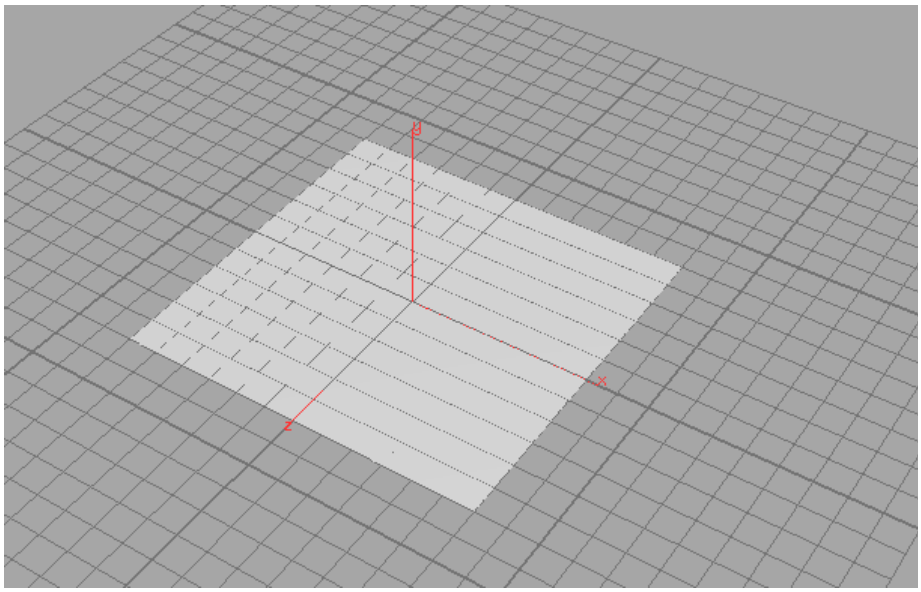


Figura 6.1 Exemple de planta

- Els perfils hauran d'estar definits dins el pla XY. Si l'usuari incorpora un perfil que es defineixi dins les 3 dimensions, el programa eliminarà la component Z convertint el perfil introduït en un que només estigui definit dins el pla XY. Veure Figura 6.2.

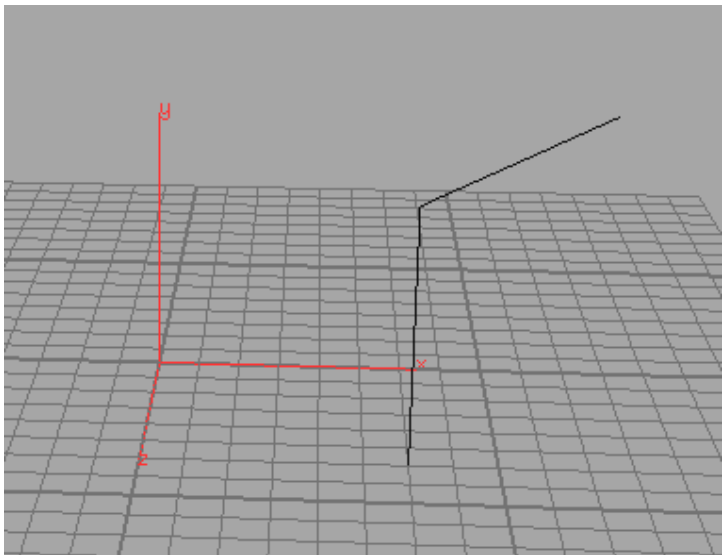


Figura 6.2 Exemple de perfil

A la Figura 6.2 es pot observar com s'ha definit un perfil. Primer de tot ha d'estar al pla XY, això vol dir que la component Z dels vèrtexs que formen el perfil sempre ha de ser igual. Una altra cosa a tenir en compte és la posició on està definit el perfil, ja que influirà a quina aresta s'associa aquest perfil: cada aresta s'associa al perfil més proper.

- Un cop definida la planta i el perfil es mostra el resultat a la Figura 6.3. Com que només s'ha definit un perfil, cada costat de l'edifici busca el perfil que està més aprop d'ell, per tant en aquest cas s'associa el mateix perfil en tots els costats.

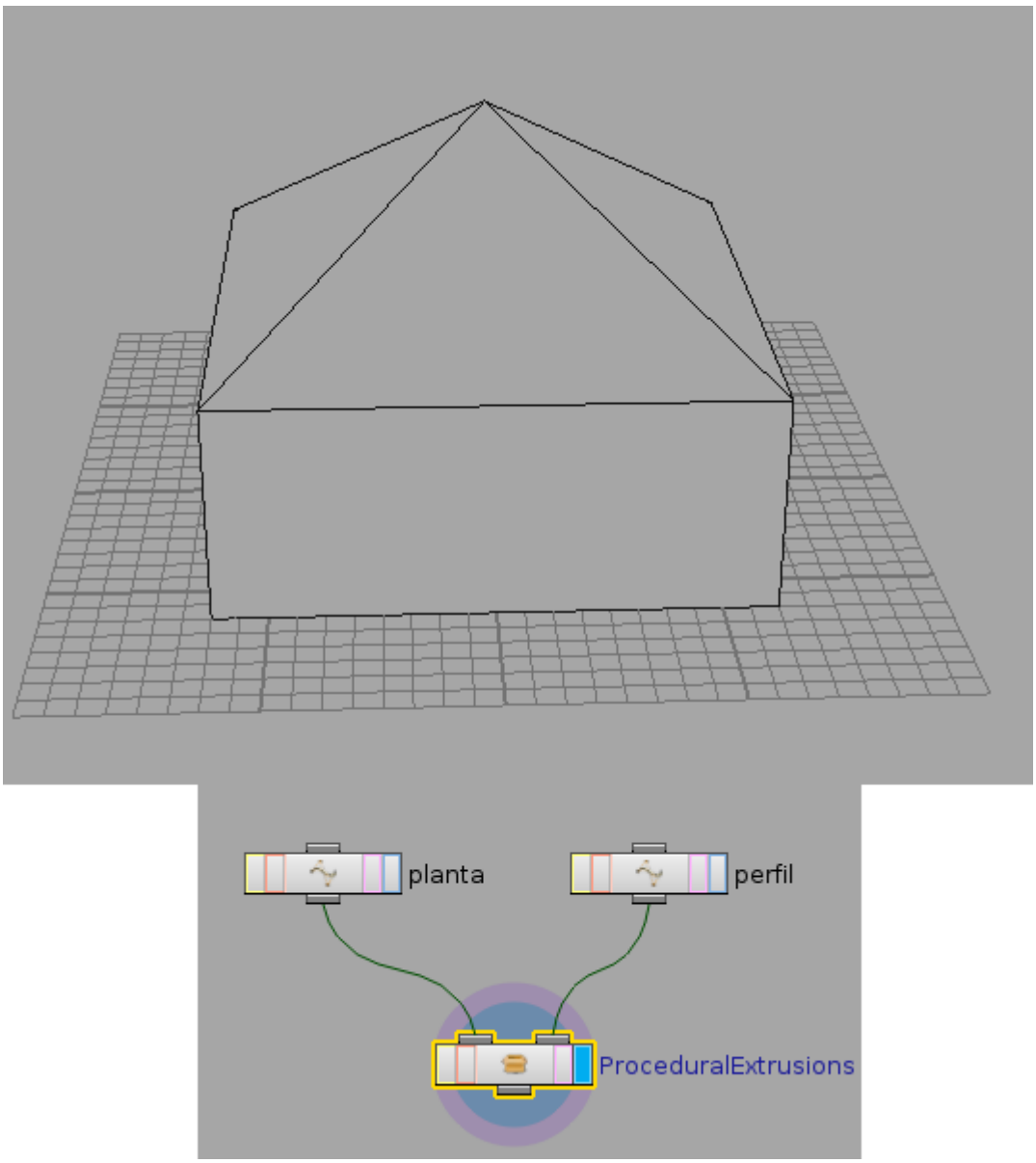


Figura 6.3 Exemple de resultat d'un edifici

Observant la Figura 6.3 veiem que s'han utilitzat la planta i el perfil que s'havien definit anteriorment a les Figures 6.1 i 6.2, per tant l'edifici resultant ha quedat de forma quadrada. Al tenir només definit un perfil, tots els costats de l'edifici tenen la mateixa forma, creixent totalment vertical i llavors es dirigeixen cap al centre amb una certa inclinació. A la Figura, es pot observar a dalt el resultat, i a baix es mostra com quedarien els nodes a la zona d'arbre de nodes de Houdini.

En el cas que es volgués definir més d'un perfil, existeixen 2 maneres de treballar: la més simple és anar creant tants nodes “curve” com perfils es vulguin definir. L'altre manera és

definir un sol node “curve” i seguit d'aquest se li enllacen nodes “transform” de manera que la sortida de cada node “transform” és un nou perfil.

Per associar els perfils a les arestes de la planta es farà mitjançant distàncies, o sigui per cada aresta de la planta es buscarà el perfil que estigui més a prop de tots els que l'usuari hagi definit.

A les Figures 6.4 i 6.5 es pot veure com quedaria un edifici definint 2 perfils diferents.

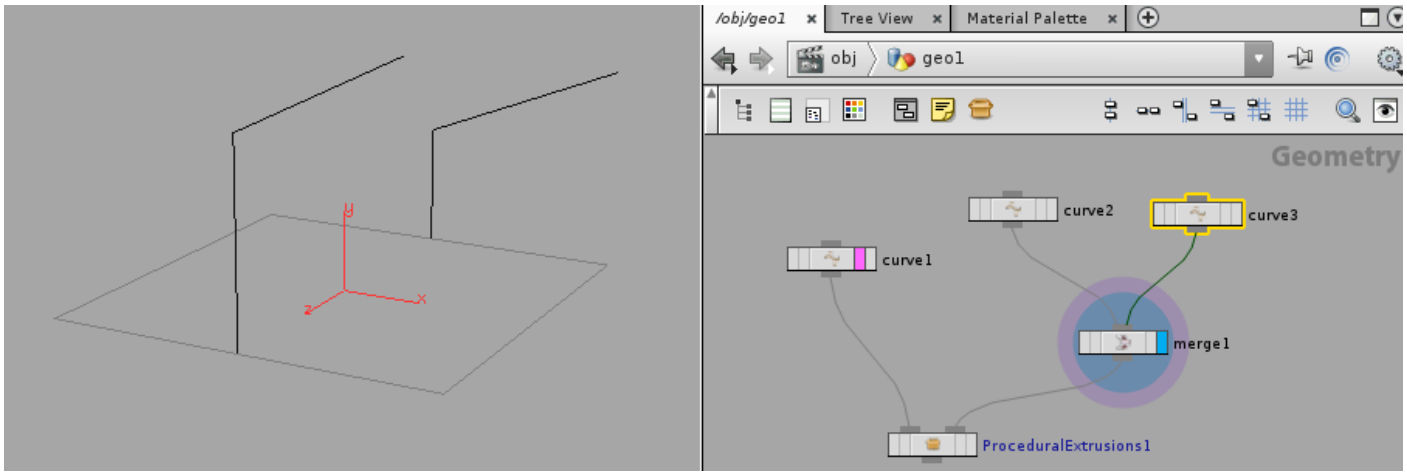


Figura 6.4 Definició de més d'un perfil

Tal i com es veu en la Figura 6.4, per definir 2 perfils, s'han creat 2 nodes curve, i llavors s'han unit amb el “merge” per connectar-ho amb el node de ProceduralExtrusions.

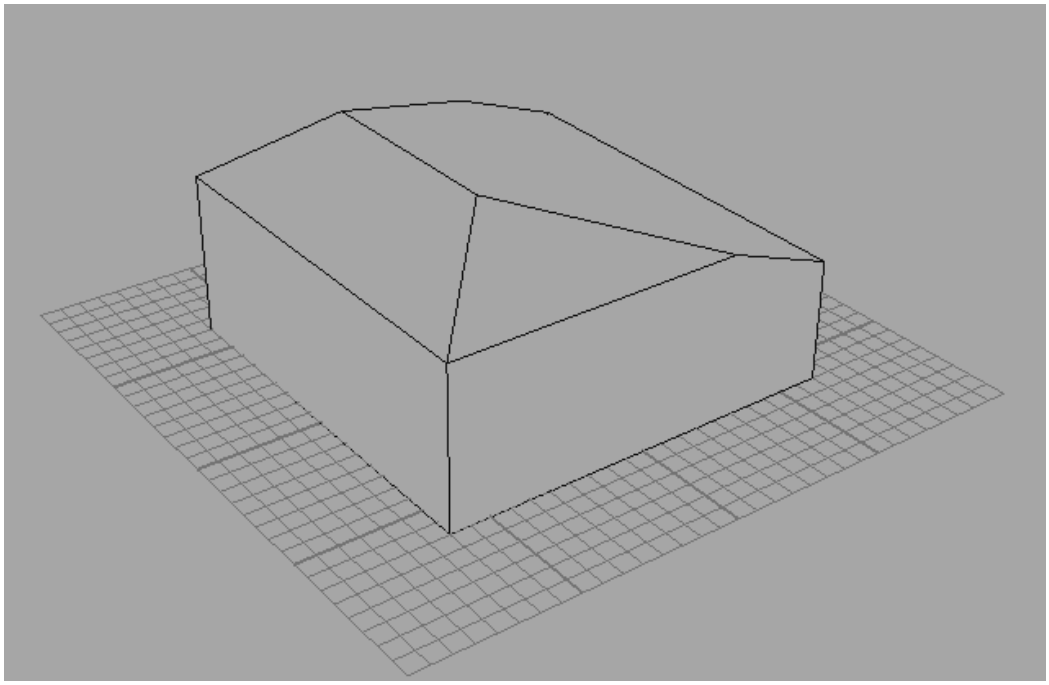


Figura 6.5 Exemple resultat edifici amb 2 perfils

Com a precondició del sistema, es requereix que s'hagi definit una planta i un perfil com a mínim.

6.2. Requisits no funcionals

- El programa ha de ser eficient: no fer càlculs redundants ni utilitzar més memòria de la necessària.
- Permetre ser extensible: que es puguin incorporar innovacions d'algorismes a mesura que es vagin desenvolupant.
- La interfície ha de ser intuïtiva i usable.
- Realitzar una documentació sobre el funcionament bàsic del programa.
- Hauria de poder funcionar en diversos sistemes operatius.

Com que la plataforma on es desenvolupa és el Houdini, els requeriments mínims són els que requereix el Houdini per poder funcionar, el qual és un software multi-plataforma permetent executar-lo per GNU/Linux, Microsoft windows o Mac OS X.

Cal destacar que la llibreria que s'utilitza en el projecte, JPype compleix els requeriments ja que és multi-plataforma.

7. Estudis i decisions

En aquest capítol es llisten i detallen les eines utilitzades per a dur a terme el desenvolupament del projecte.

7.1. Houdini

El Houdini és una eina per a la creació avançada d'efectes visuals 3D, que permet controlar l'animació des de la mateixa interfície o utilitzant un dels 2 llenguatges d'script que incorpora. El més antic és el Hsript. És exclusiu de Houdini i existent des de les primeres versions. L'altre 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.

Houdini cobreix totes les principals regions de producció en 3D incloent:

- Modelatge: totes les entitats de geometria estàndard, incloent polígons.
- Animació: animació de fotogrames, manipulació del canal i captures de moviment.
- Partícules.
- Dinàmiques: Dinàmica de cossos rígids, de fluids computacionals, corbes dinàmiques, etc.
- Il·luminació.
- Rendering.
- Volumètrica: generació, població, manipulació, d'escalars i vectors.
- Composició: compositor complet de punt flotant en profunditat d'imatges(capes)
- Plugin per al desenvolupament: biblioteques per a la extensibilitat de l'usuari.

Houdini és un medi obert i suporta una varietat de seqüències de comandes d'API. Python és cada vegada més el llenguatge de scripting de la tria del paquet, i està destinat a substituir el llenguatge original Hscript.

El 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.

L'entorn del Houdini està compost de la següent manera:

- Barra d'eines, a la part superior. On es generen totes les figures i accions.
- Zona de treball. Espai on es poden visualitzar els resultats utilitzant diverses vistes, i es poden seleccionar o modificar els objectes que hi hagi dins l'escena.
- Arbre de nodes, a la part inferior dreta. Zona on es representen tots els objectes de l'escena de forma independent enllaçades les unes amb les altres.
- Zona d'interfície, a sobre de l'arbre de nodes. Zona que permet consultar o modificar les propietats del node que s'hagi seleccionat al arbre de nodes.

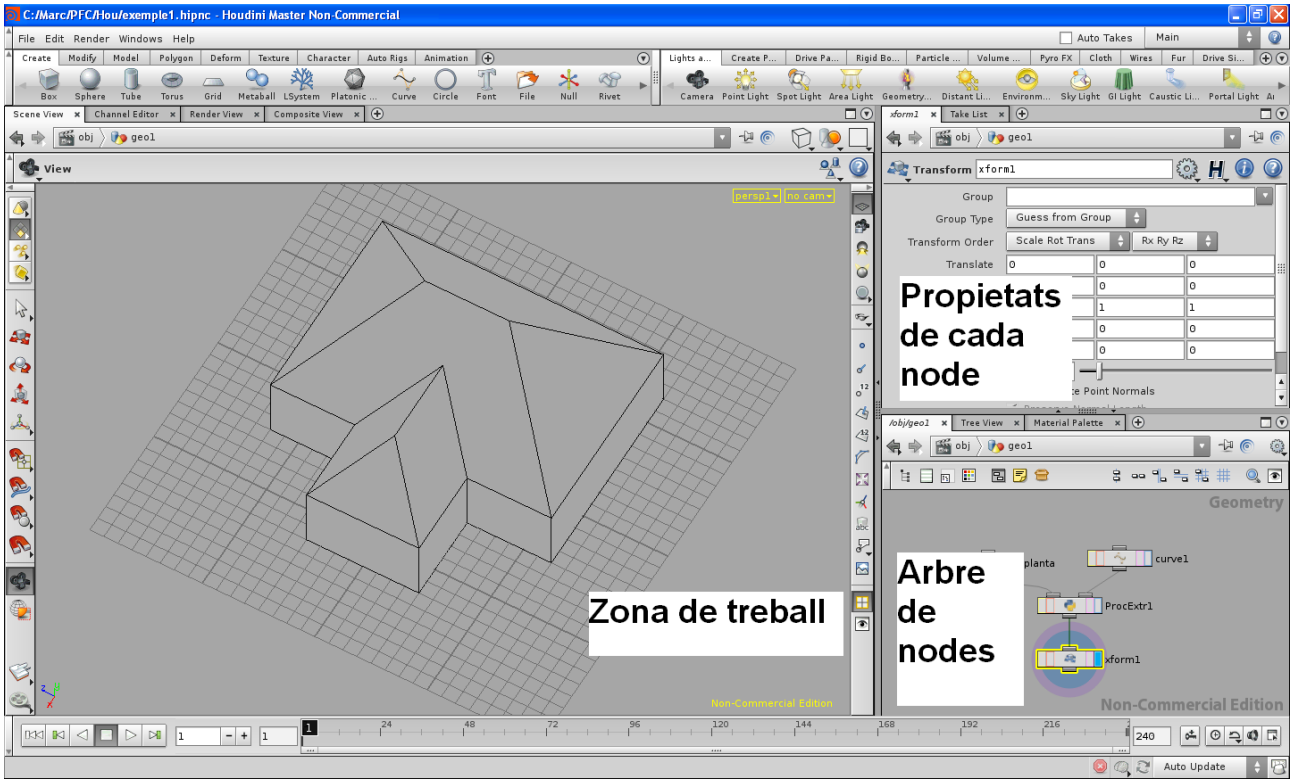


Figura 7.1.1 Entorn de treball del Houdini

7.1.1. Nodes Houdini

Per crear un node Houdini cal polsar la tecla tabulador tenint el cursor a la zona d'arbre de nodes, llavors apareixerà una finestra amb tots els nodes disponibles, organitzats per famílies.

Si d'entrada ja es sap quin node crear, es pot escriure directament el nom en aquesta finestra i el Houdini anirà filtrant tots els nodes pel que s'estigui escrivint, en la Figura 7.1.2 es veu com s'ha buscat “sphere”.

Figura 7.1.2 Menú tabulador de nodes de Houdini

A les figures 7.1.3 i 7.1.4 es veuen els resultats al Houdini. D'una banda es veu ja la representació de l'esfera a la zona de treball, i a més també es genera el node a la zona corresponent.

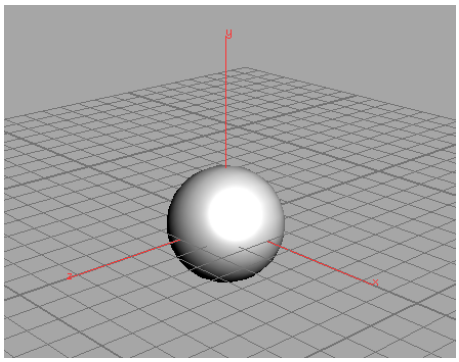


Figura 7.1.3 Resultat a la zona de treball

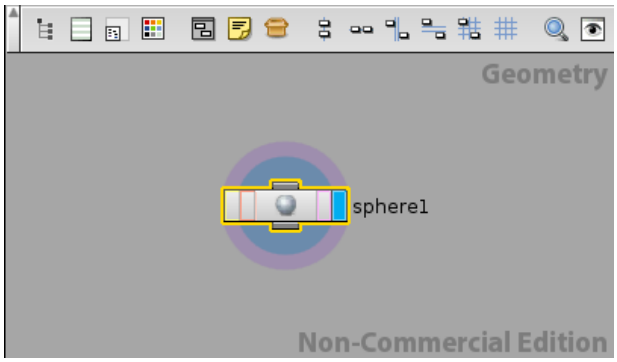


Figura 7.1.4 Resultat a la zona arbre de nodes

7.1.2. Estructura de dades de la geometria de Houdini

Houdini organitza l'estructura de dades en objectes, primitives, vèrtexs i punts.

Un objecte és tot el conjunt de geometries que el formen. Pot ser un edifici, una porta, un cub, una esfera, etc. Aquest objecte estarà format per primitives.

Una primitiva és anomenada així per la seva constitució en les parts que la formen. Les primitives més típiques són les de un cercle, un triangle o un quadrat. O, si parlem en 3 dimensions, una esfera, un tub, un cilindre, etc. Es pot donar el cas que es tingui un objecte que sigui un triangle, llavors aquest objecte només estaria format per una primitiva que seria la d'un triangle.

A continuació, cada primitiva està formada per vèrtexs depenent de la primitiva. Per exemple un quadrat està format per 4 vèrtexs.

I finalment un vèrtex està format per un punt. Aquest punt ja té les dades suficients per representar-se en l'espai 3D: les coordenades X,Y i Z i el color RGB, a més d'altres dades que l'usuari pugui haver afegit.

En la Figura 7.1. es mostra l'esquema de l'entitat relació de l'estructura de dades.

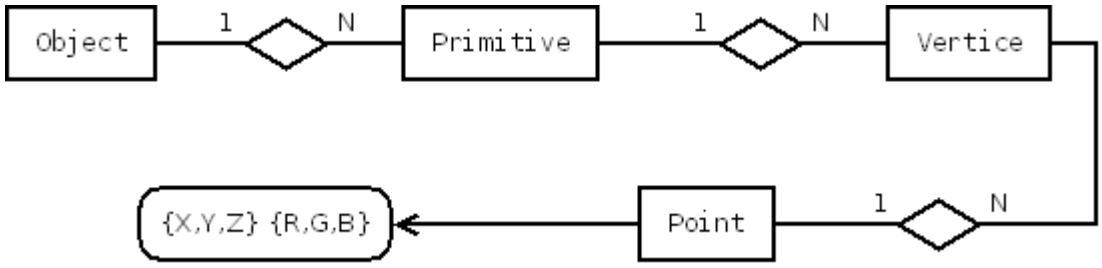


Figura 7.1.5 Esquema entitat relació de l'estructura de dades del Houdini

Tal i com es pot apreciar veient l'esquema, un objecte està format per una o més primitives, però una primitiva només pot pertànyer a un objecte. El mateix passa amb la relació que hi ha entre primitives i vèrtexs: una primitiva està formada per un o més vèrtexs, però un vèrtex només pot pertànyer a una primitiva.

En canvi en la relació entre vèrtexs i punts és al revés: és el punt que pot pertànyer a un o més vèrtexs, i el vèrtex només pot pertànyer a un punt. En la Figura 7.1.6 es veu aquesta última relació més detallada.

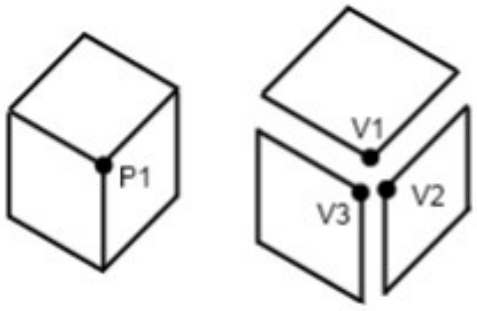


Figura 7.1.6 Relació entre vèrtex i punt

Com es veu en la Figura 7.1.6 el punt P1 està correspost als vèrtexs V1,V2 i V3, i cada vèrtex està relacionat únicament amb el punt P1

7.1.3. Nodes bàsics de Houdini

En aquest apartat s'explicaran alguns dels nodes bàsics propis del Houdini, que s'hagin utilitzat en aquest projecte. Primer de tot cal indicar que per visualitzar un node se li ha de clicar el rectangle de més a la dreta del node de color blau (en la zona de l'arbre de nodes), i un cop queda remarcat es visualitzarà el node a la zona de treball. Veure Figura 7.1.7.

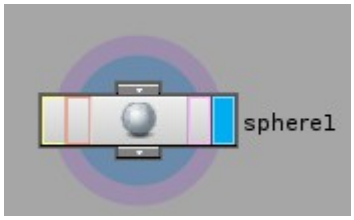


Figura 7.1.7 Node seleccionat

7.1.3.1. Curve

Node que serveix per dibuixar una corba o polígons, i editar-la de la forma desitjada. Aquest node serà important perquè s'utilitzarà per poder dibuixar la planta i perfils que tindrà l'edifici. Veure la Figura 7.1.8.

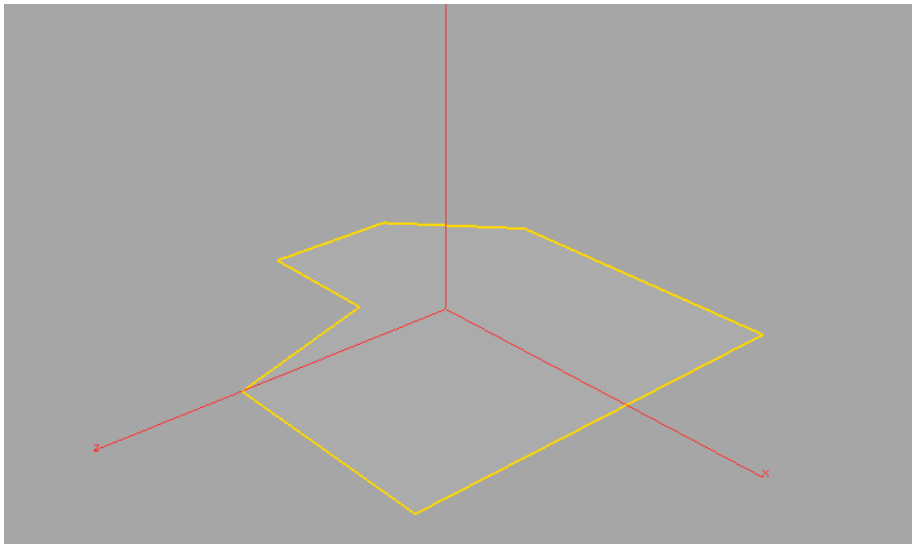


Figura 7.1.8 Polígon creat amb el node curve

7.1.3.2. Merge

El Merge permet ajuntar més d'un node a la vegada per poder-los visualitzar tots junts. Aquest node permetrà ajuntar els perfils que s'introdueixin com a paràmetres per a crear l'edifici. D'aquesta manera el node que es crearà pel projecte rebrà únicament 2 entrades: planta i perfils. Sense el merge, es necessitaria un número d'entrades variable depenent del número de perfils que es volgués introduir. Veure Figura 7.1.9. També servirà per poder visualitzar més parts de l'edifici juntes, quan s'integri amb el buildingEngine.

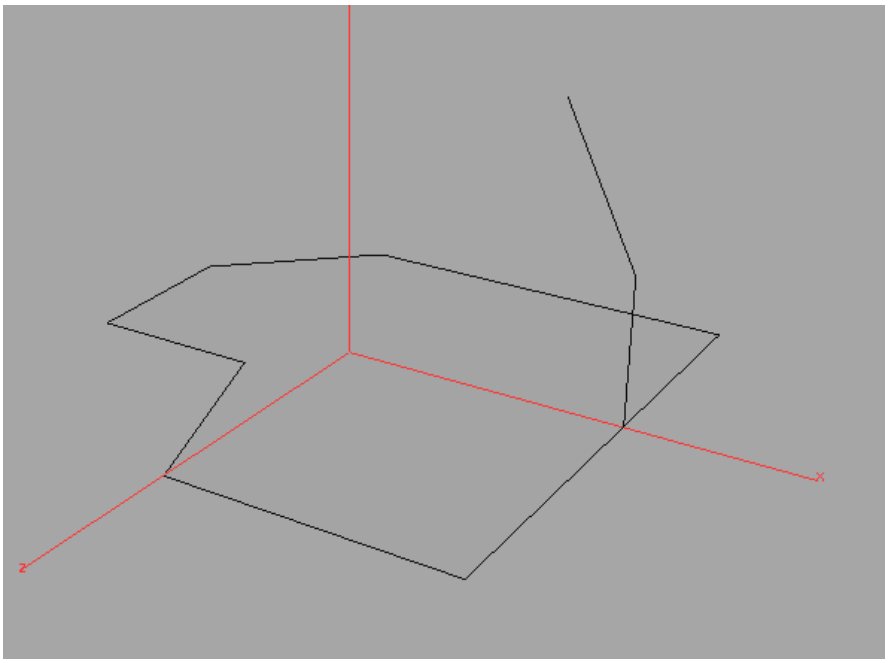


Figura 7.1.9 El merge permet visualitzar les 2 curves simultàniament (planta i perfil)

7.1.3.3. Delete

Aquest node serveix per poder esborrar geometria. De manera que serà útil per esborrar una part de la Figura que s'estigui veient, o també al revés, esborrar-ho tot excepte aquesta part. Veiem un exemple del seu funcionament a la Figura 7.1.10.

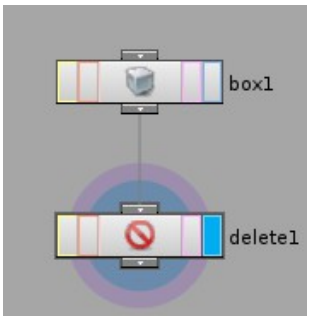


Figura 7.1.10 Node delete connectat amb box

S'ha connectat la sortida del box amb l'entrada del delete, així doncs ara modificant els paràmetres del delete es podrà esborrar alguna cara de la caixa. Veure Figura 7.1.11.

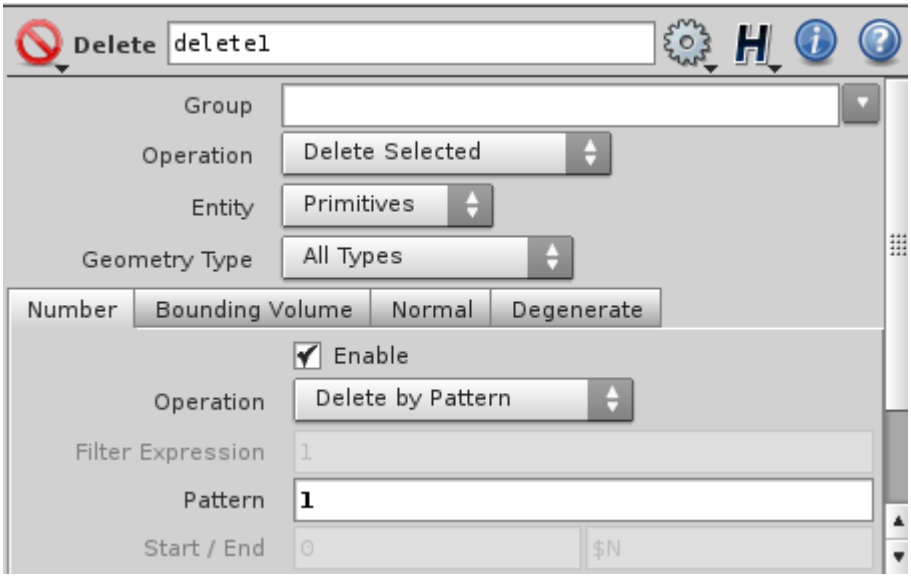


Figura 7.1.11 Paràmetres del node delete

El més important és la casella “operation” de a dalt. En la Figura 7.1.11 està seleccionat “Delete Selected”. Per tant el node esborrarà únicament el que es seleccioni. L'altre opció seria escollir “Delete Non-Selected” i faria la operació al revés esborrar-ho tot excepte el que estigui seleccionat.

Llavors hi ha una altre casella operation per indicar com esborrar la geometria, en aquest cas està “delete by pattern”, o sigui que s'eliminaran les primitives que seleccionem.

Per acabar, a la casella “Pattern” s'ha d'indicar quina primitiva esborrar. En l'exemple de la Figura 7.1.11 s'ha indicat la 1. El resultat es pot veure a la Figura 7.1.12. Esborrant la primitiva 1, s'ha esborrat aquesta cara com es pot veure.

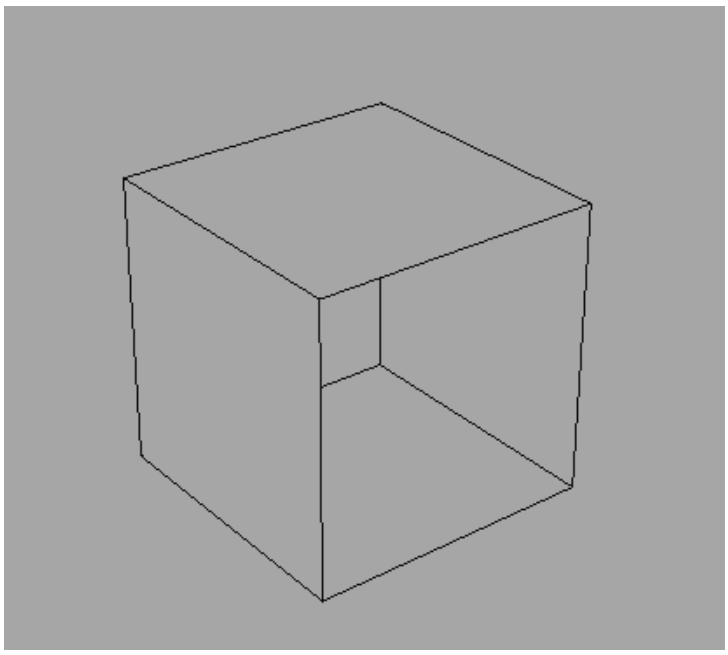


Figura 7.1.12 Resultat del node Delete de l'exemple explicat

A la Figura 7.1.13 es pot veure el mateix exemple, però executant l'operació amb “Delete Non-Selected”. Tal i com es veu, el node delete ha esborrat tot el cub excepte la primitiva que s'havia indicat, la 1.

El node delete és útil per esbrinar si la Figura s'està dibuixant correctament, es poden anar esborrant primitives de mica en mica per tal de veure si hi ha alguna incoherència.

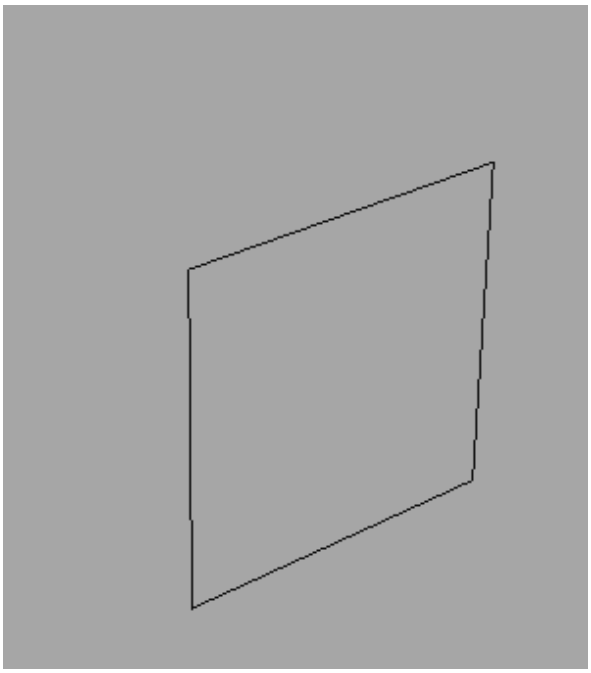


Figura 7.1.13 Delete amb operació Delete Non-Selected

7.2. Python

Python és un llenguatge de programació creat per Guido van Rossum 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. Per aquest projecte s'ha emprat la versió 2.6, que es el que porta el Houdini.

Python és considerat com la "oposició lleial" a Perl, llenguatge amb el qual manté una rivalitat amistosa. Els usuaris de Python considerem 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, wxWidgets(wxPython), PMW, 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.

El Python permet diversos paradigmes de programació (és multi paradigma), com poden ser orientat a objectes, imperatiu, i, en menor part, programació funcional, procedural i reflexiu. Una de les coses que incorpora la programació funcional del Python és la possibilitat de crear funcions lambda, que és un tipus d'expressió que permet avaluar termes.

7.2.1. Python dins el Houdini

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. Veure Figura 7.2.1.

Usant la consola es poden fer totes les accions disponibles a la interfície gràfica, però de manera més automàtica i independent per a l'usuari.

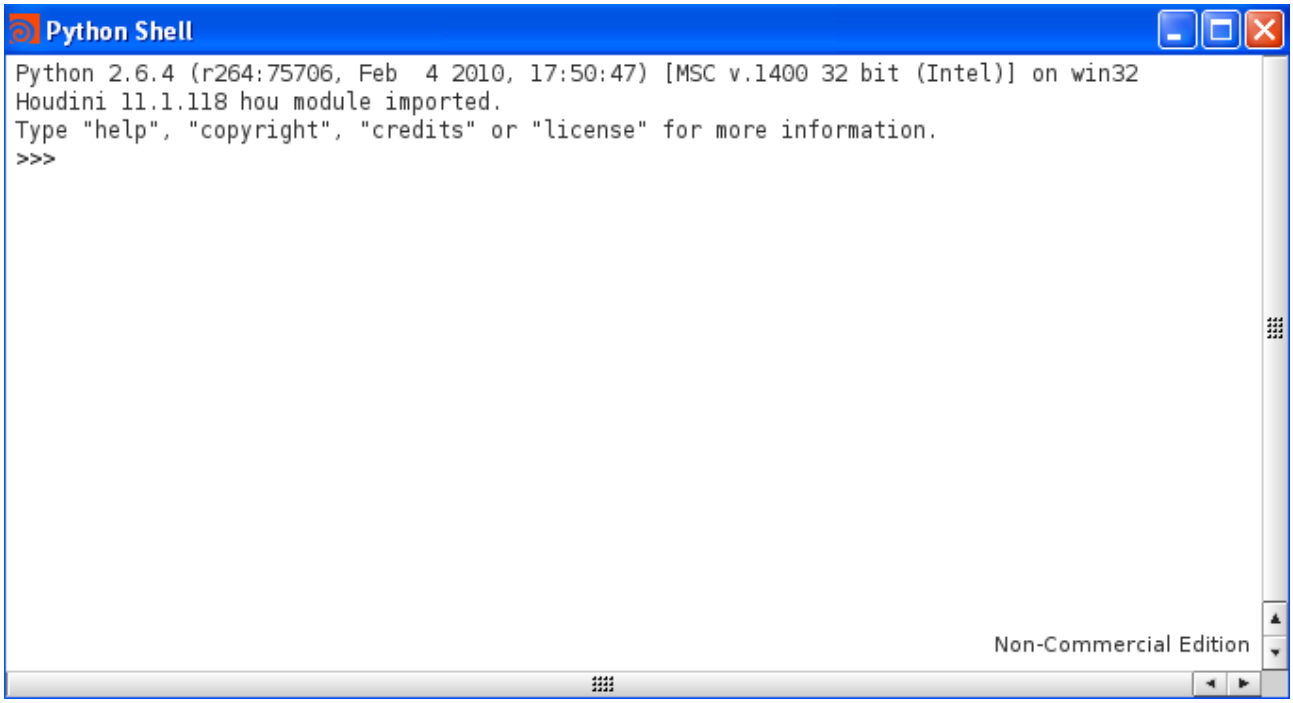


Figura 7.2.1 La consola Python del Houdini

Com que Python és un llenguatge interpretat, es poden executar funcions en temps real, tal i com es faria prement un botó del programa. Veure Figura 7.2.2.

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). A la Figura 7.2.2 podem veure un exemple creant una esfera amb la consola de Python.

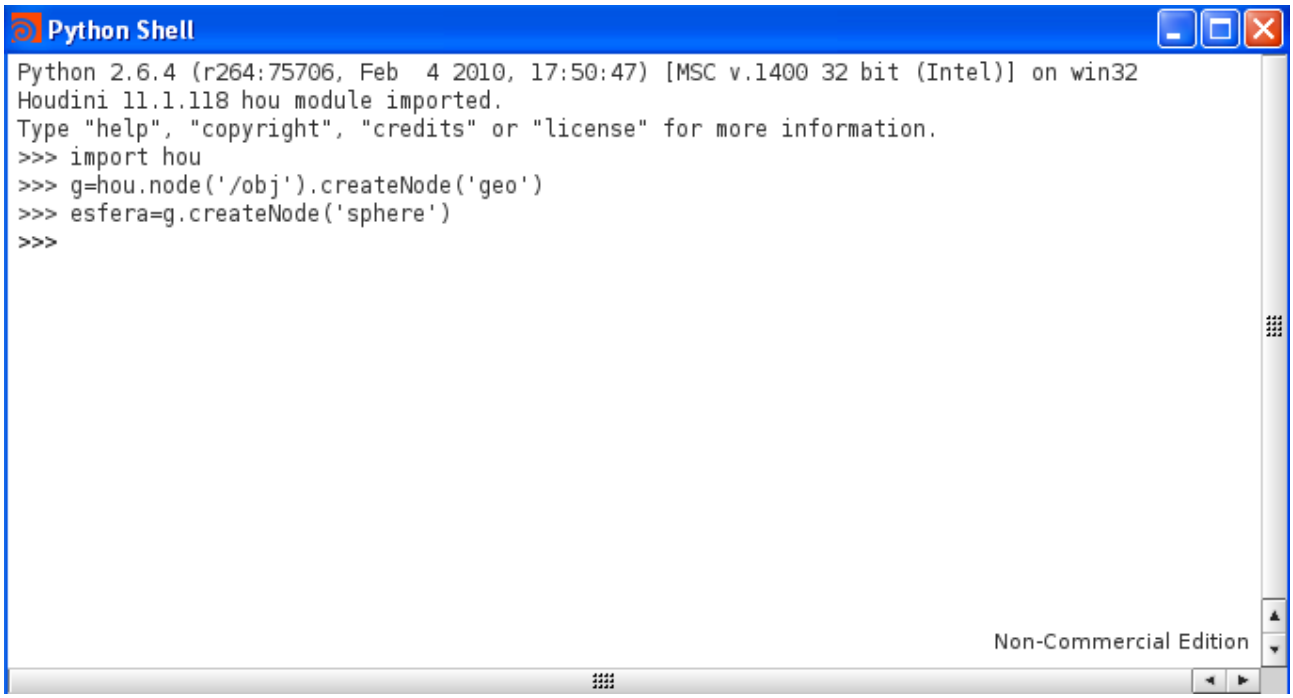


Figura 7.2.2 Creant una esfera des de la consola Python de Houdini

Primer de tot s'importa el mòdul hou. Seguidament es crea un objecte de tipus 'geo' que emmagatzemarà tots els objectes. Finalment es crea l'esfera 'sphere' utilitzant la sentència "createNode()".

El resultat és exactament el mateix que el que s'ha mostrat d'exemple en el capítol 7.1.1 Nodes Houdini. O sigui en la zona de treball es dibuixarà la representació de l'esfera, i en la zona de l'arbre de nodes es crearà aquest node nou sphere.

7.3. BuildingEngine

Per a la creació d'edificis virtuals s'ha utilitzat la llibreria buildingEngine, que és un mòdul de **skyline**Engine. El buildingEngine permet la definició i la construcció procedural d'edificis i altres estructures arquitectòniques.

A la Figura 7.3.1 es pot veure un exemple del funcionament del buildingEngine.

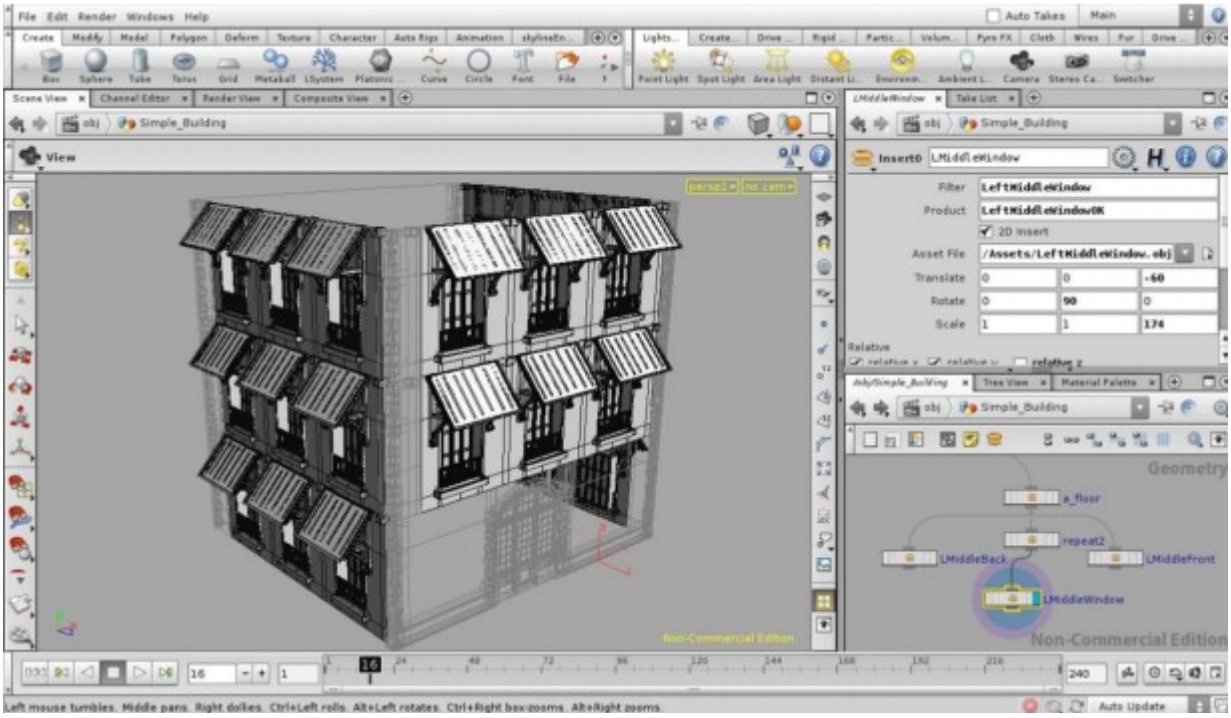


Figura 7.3.1 Exemple de funcionament del buildingEngine

7.3.1. Nodes de buildingEngine

Quan es treballa amb el buildingEngine tota la geometria es descriu mitjançant etiquetes semàntiques. Per exemple, poden haver-hi etiquetes com “facade” o “door” que representarien la part de l'edifici façana o porta.

Els nodes del buildingEngine generen els edificis a través d'aquestes etiquetes utilitzant dos atributs: “filter” i “product”. El “filter” ens indica la geometria que processarà el node, mentre que el “product” ens indica l'etiqueta de la geometria que es generarà a partir d'aquest.

A la Figura 7.3.2 es mostra com queda representat un node de buildingEngine a dins el Houdini.

Figura 7.3.2 Exemple d'un node de buildingEngine

7.3.1.1. Node Comp

El node Comp divideix un model de volum per les seves components. Aquest node classifica la geometria entrant d'acord amb els vectors normals dels polígons, com podria ser la façana, les altres cares i el sostre.

Per exemple es crearà un prisma i a partir d'aquest s'utilitzarà el node Comp per a obtenir les seves components per separat. Veure Figura 7.3.3.

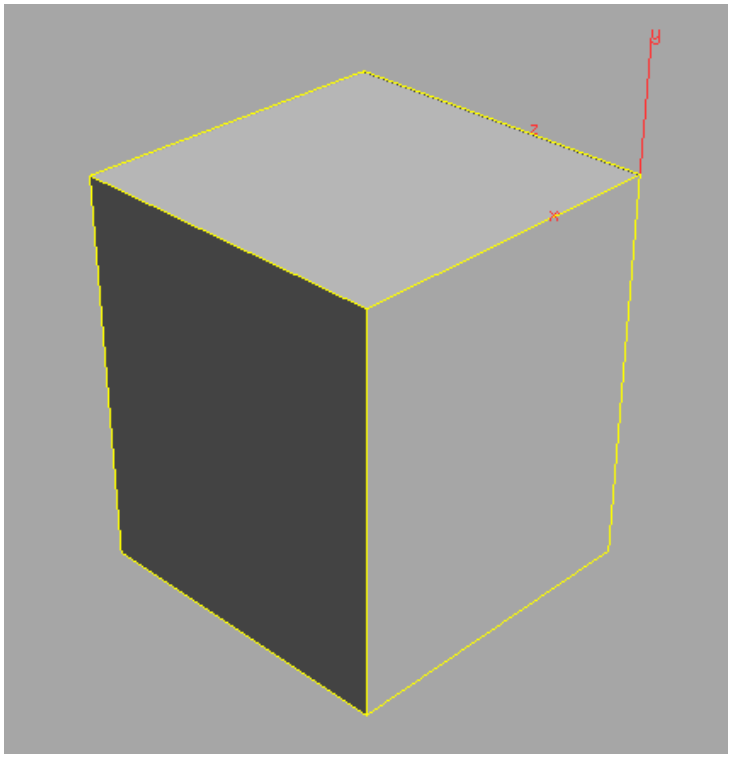


Figura 7.3.3 Exemple d'un prisma

Per seleccionar les parts que formen el prisma, el node Comp té una llista de components per anar seleccionant. És l'usuari el que tria quantes parts vol. Per afegir-ne ha d'apretar en el “+” i si vol treure una selecció, la “X”. En l'exemple de la Figura 7.3.4 s'han agafat 2 components, “front” i “back”. Per seleccionar quina component es vol escollir, hi ha el paràmetre “component selection” on es pot triar una opció de les següents: front, back, left, right, top, bottom, side, all. Al costat de cada “component selection” hi ha un text per escriure-hi, doncs això serveix per a poder fer una etiqueta semàntica per a cada selecció.

Seguint en l'exemple, com que s'han escollit front i back, només es representaran les cares que estan a davant i a darrera, i l'eix que representa aquesta distància és el Z, per tant es veuran les 2 cares del prisma que estan perpendicularment a l'eix Z. Veure Figura 7.3.4.

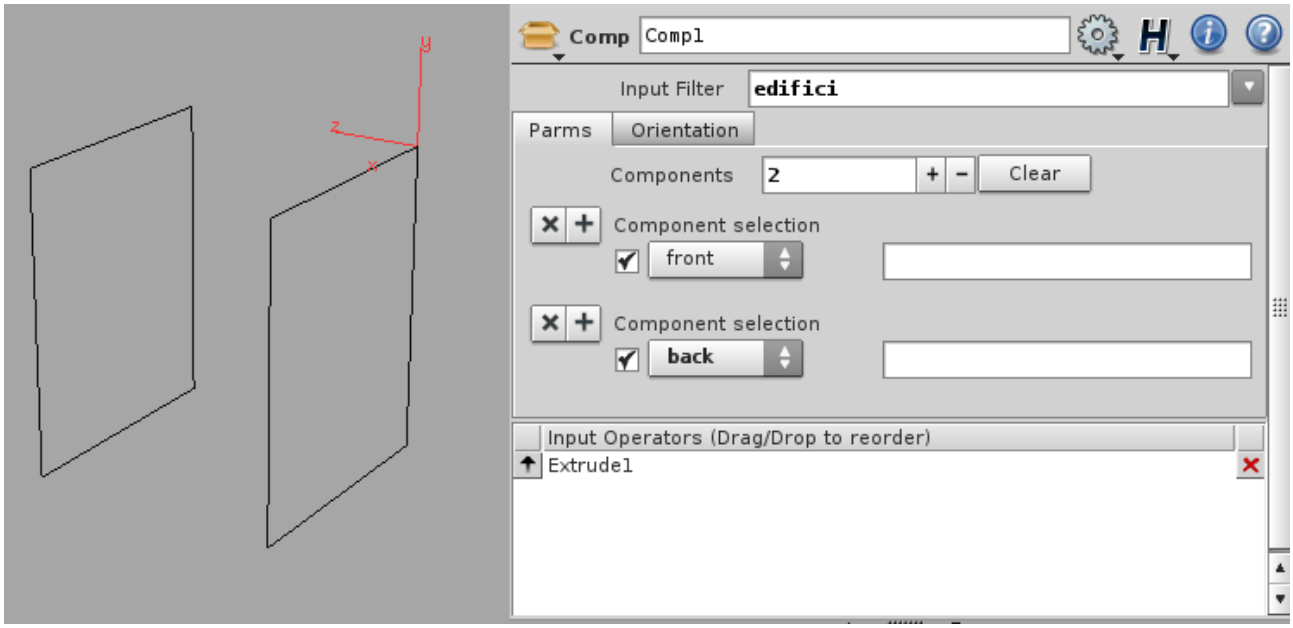


Figura 7.3.4 Resultat d'utilitzar el node Comp

7.3.1.2. Node Subdiv

El node Subdiv realitza una subdivisió de l'element que rep en elements més petits. Té tres modes de funcionament: en X, Y i Z, que permet dividir seguint cada un dels eixos.

Per veure el seu funcionament s'ha generat un quadrat d'exemple, i el node Subdiv el dividirà en 4 parts seguint l'eix Y, iguals. Veure Figura 7.3.5.

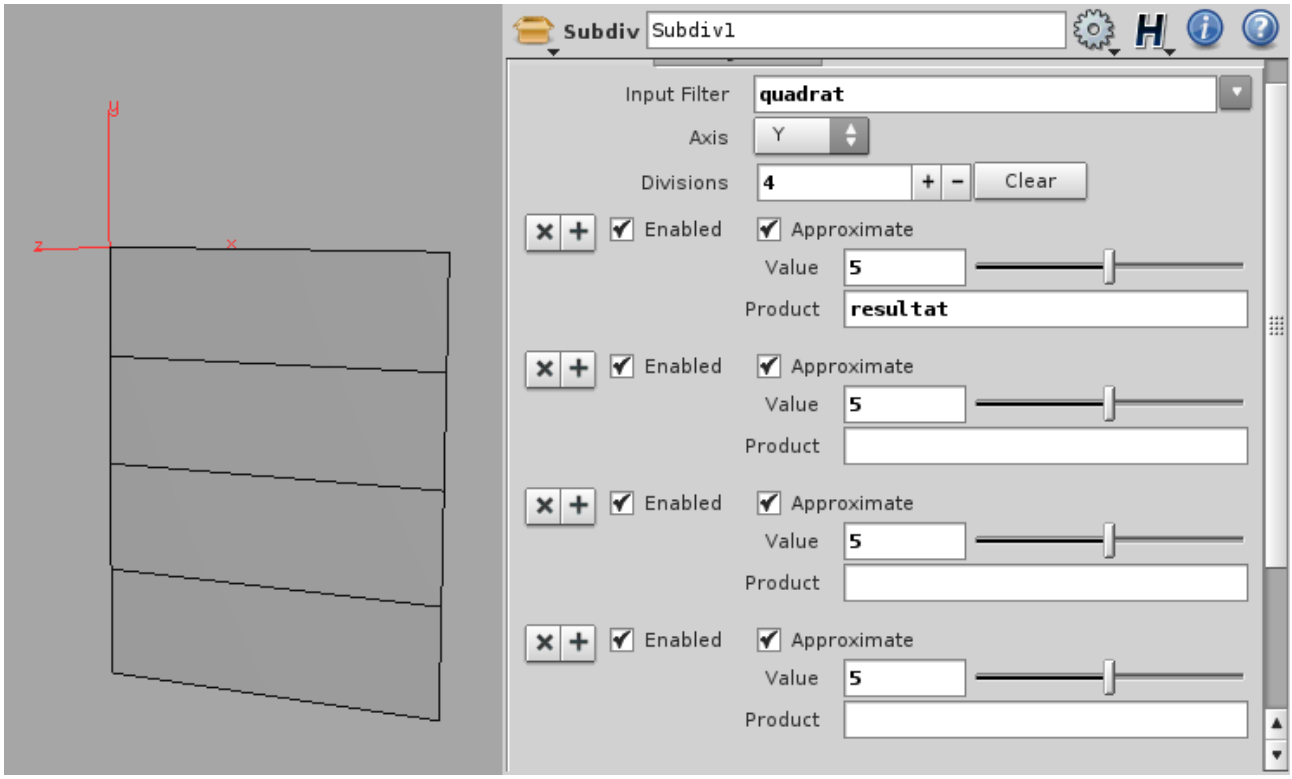


Figura 7.3.5 Exemple del funcionament del node subdiv

Funciona més o menys igual que el node Comp, l'usuari tria quantes divisions vol fer polsant la “X” o el “+” i el mode de divisió, que pot ser en X, Y o Z. Llavors cada divisió té un valor, aquest número és relatiu. Per exemple si es tinguessin 2 divisions, en una el valor 1 i l'altre un 2, voldria dir, que el quadrat es divideix en 2 parts, i una d'aquestes és el doble de gran que l'altre. Per fer parts iguals, només cal posar el mateix número en totes les divisions.

7.3.1.3. Node Repeat

El node Repeat realitza una repetició de divisions de l'element d'entrada en elements més petits. Igual que el Subdiv, el Repeat també té tres modes de funcionament: en X, Y i Z.

A la Figura 7.3.6 podem veure com funciona el mode en Y.

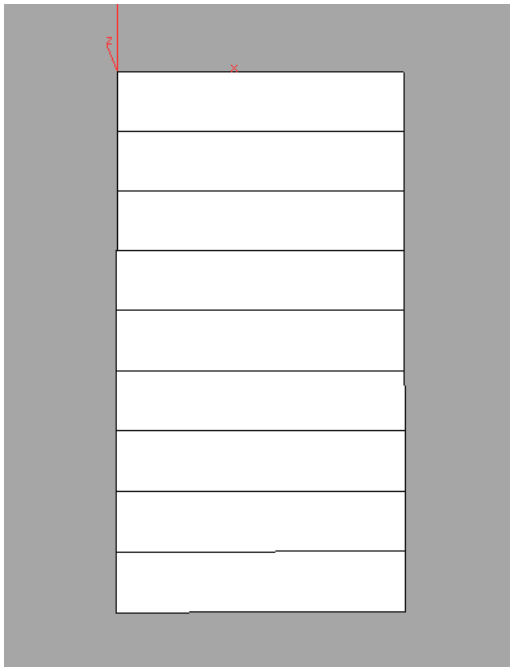


Figura 7.3.6 Funcionament node repeat en mode Y

Com es pot veure a la Figura 7.3.6 s'estan repetint els rectangles seguint l'eix Y. Tot seguit es mostra la Figura 7.3.7 on es veu el funcionament combinat amb el mode X.

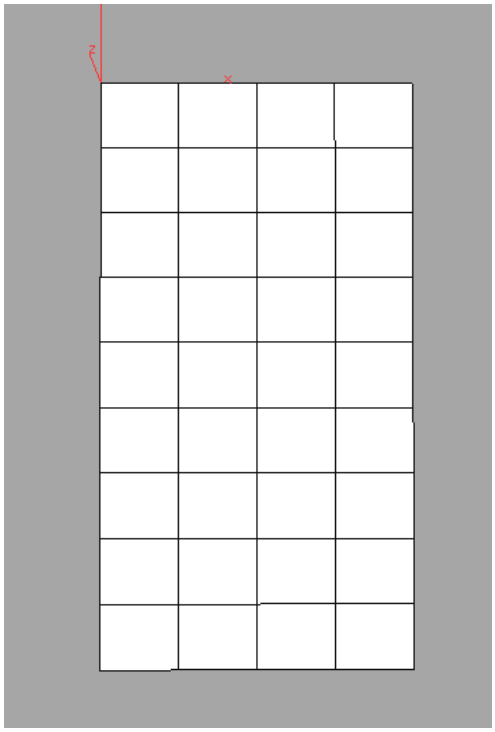


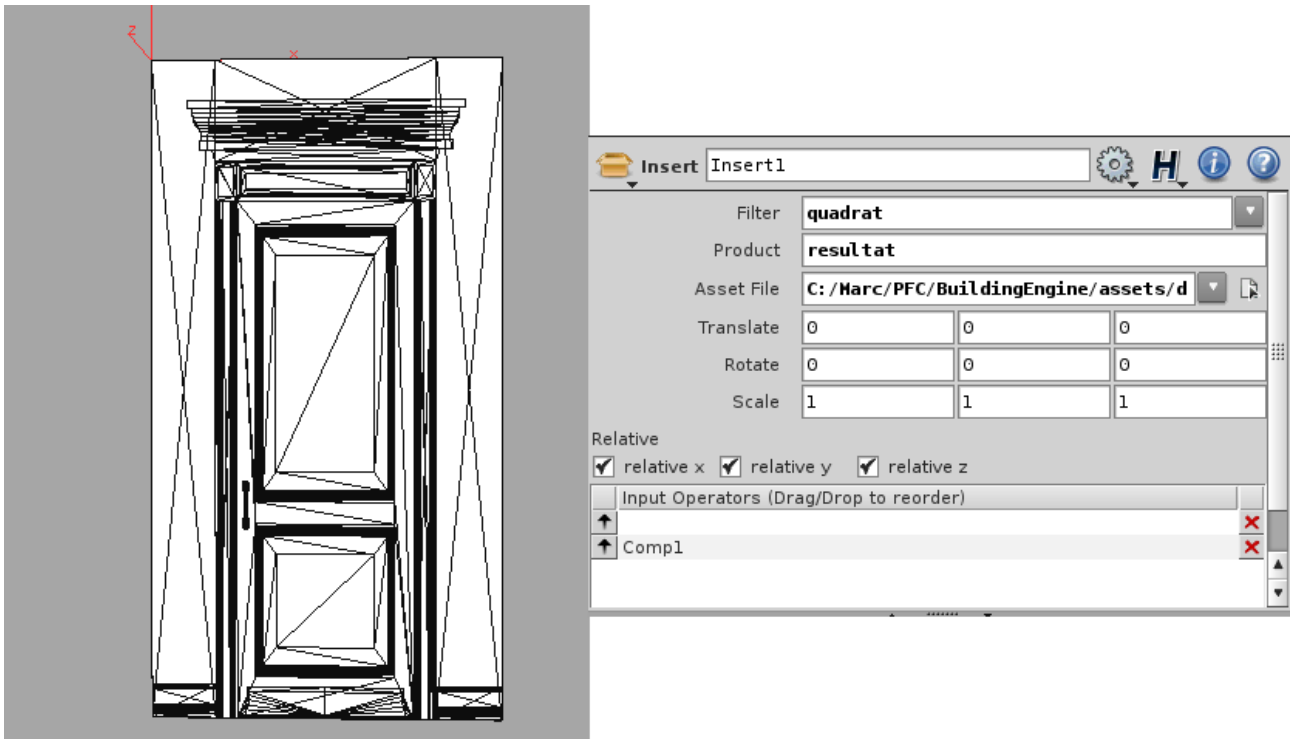
Figura 7.3.7 Exemple de funcionament del node repeat amb funcionament amb Y i X

7.3.1.4. Node Insert

El node Insert insereix una nova geometria a l'espai definit per l'element base que rep. Aquesta geometria nova l'anomenarem “asset”.

Un asset direm que és un model creat per l'artista o el modelador, i que l'ha encapsulat de tal manera que podem treballar amb ell modificant alguns paràmetres que l'artista ens permet.

En la Figura 7.3.8 es mostra un exemple d'un asset.



7.3.8 Exemple d'un asset d'una porta

A la Figura 7.3.8 es poden observar els paràmetres dels que disposa el node insert: el més important és “l'asset file”, on s'ha d'introduir la ruta de l'asset que es vol inserir, i llavors hi ha els paràmetres de translació, rotació i escalabilitat.

7.4. SitePlan

El siteplan és la llibreria original feta per Tom Kelly i Peter Wonka que utilitza els processos d'extrusió per generar edificis, i també inclou funcions per generar finestres, portes, etc.

A continuació s'explicaran les classes del codi que es reutilitzaran per fer possible el projecte.

7.4.1. Skeleton

És la classe principal que s'utilitzarà pel projecte. La classe Skeleton és la classe que té tota l'estructura de dades de la geometria resultant de l'edifici.

L'estructura de dades està definida per cares i vèrtexs, però la manera d'emmagatzemar aquesta informació és mitjançant estructures recursives. Més endavant s'explicaran aquestes estructures.

7.4.2. Plan

El Plan és l'objecte que té la informació necessària per a poder fer l'edifici, o sigui la planta i els perfils. La planta es crea mitjançant objectes Bar, que representen una façana, formada per 2 punts. La planta es defineix per l'objecte Profile, però que aquest objecte també està format per objectes Bar, de manera que la planta i els perfils tenen bàsicament la mateixa estructura de dades.

7.4.3. PlanSkeleton

El PlanSkeleton és un pas intermedi entre el Plan i el Skeleton. La manera més lògica per fer funcionar el codi seria la següent: Es crea l'objecte Plan amb la informació prèvia per a crear l'edifici i el Skeleton buit. Llavors és quan apareix el PlanSkeleton per enllaçar el Skeleton amb el Plan, i un cop enllaçat, l'objecte Skeleton ja pot crear l'edifici.

7.4.4. Estructures de dades

Com s'ha dit anteriorment, aquest codi fa servir estructures de dades recursives. Primer de tot es mostra un exemple d'aquesta estructura extreta utilitzant el codi font, ja que té una funcionalitat per extreure-la en format XML. Veure Figura 7.4.1.

```
<points serialization="custom">
  <list>
    <utils.Loop>
      <start>
        <me class="straightskeleton.ui.Bar">
          <start>
            <x>10.0</x>
            <y>210.0</y>
          </start>
          <end>
            <x>10.0</x>
            <y>10.0</y>
          </end>
        </me>
        <next>
          <me class="straightskeleton.ui.Bar">
            <start reference="../../../me/end"/>
            <end>
              <x>210.0</x>
              <y>10.0</y>
            </end>
          </me>
          <next>
            <me class="straightskeleton.ui.Bar" reference="../../me"/>
            <next>
              <me class="straightskeleton.ui.Bar">
                <start reference="../../../../me/end"/>
                <end>
                  <x>210.0</x>
                  <y>210.0</y>
                </end>
              </me>
              <next>
                <me class="straightskeleton.ui.Bar">
                  <start reference="../../../me/end"/>
                  <end reference="../../../../../../me/start"/>
                </me>
                <next reference="../../../../../"/>
                <prev reference="../.."/>
              </next>
              <prev reference="../.."/>
            </next>
            <prev reference="../.."/>
          </next>
          <prev reference="../.."/>
        </next>
        <prev reference="../next/next/next/next"/>
      </start>
    </utils.Loop>
  </list>
</points>
```

Figura 7.4.1 Exemple d'estructura de dades de SitePlan

La Figura 7.4.1 és una extracció de l'estructura de dades que fa servir el SitePlan, en la Figura s'han eliminat detalls innecessaris, per així mostrar únicament la informació important, que s'utilitza dins el projecte. A la Figura només es mostra l'estructura de la planta.

Primer de tot cal explicar que la planta que s'està mostrant en la Figura 7.4.1 és quadrada, per tant s'haurien d'observar 4 vèrtexs.

Començant a llegir el codi xml, primer es defineix l'objecte Loop que es tracta d'una classe recursiva, dins d'aquesta hi ha tota la informació corresponent a la planta. Un cop definit el Loop, es defineix l'objecte Bar, per definir la primera façana, tal i com es veu comença amb l'etiqueta <start> al punt (10.0,210.0) i l'etiqueta <end> indica on acaba, en aquest cas en (10.0,10.0). A continuació es defineix un altre Bar per continuar formant la planta, però ara no indica on comença ja que s'aprofita amb l'estructura recursiva que el punt d'inici d'aquest Bar és l'últim de l'anterior, per tant només s'indica l'etiqueta <end> al punt (210.0, 10.0). D'aquesta manera aquest segon Bar que s'ha definit comença al punt (10.0, 10.0) i acaba en (210.0, 10.0).

Així successivament es pot anar recorrent l'estructura fins que arribi al punt des d'on s'ha començat definint la planta. En el cas dels perfils es defineixen igual, però el Bar final del perfil no acabarà en el punt d'origen del perfil.

A continuació es mostra el diagrama de classes del mòdul CampSkeleton, (només mostrant les classes que interessen. Veure Figura 7.4.2.

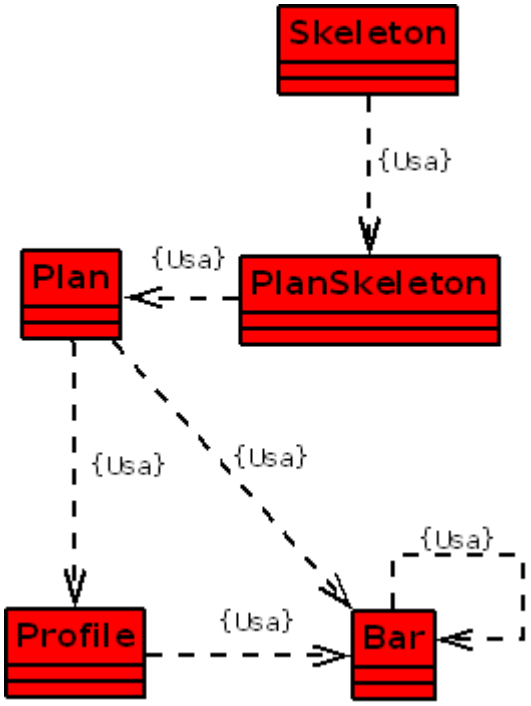


Figura 7.4.2 Diagrama de classes del mòdul de CampSkeleton

7.5. JPype

JPype és una eina per tal que programes fets en Python tinguin un accés total a classes o llibreries implementades en Java.

Per executar qualsevol codi en Java al nostre ordinador, necessitem tenir el Java instal·lat. Això vol dir instal·lar una màquina virtual que serà l'encarregada de l'execució dels codis Java.

Per tant el JPype el què fa és interconnectar el Python directament amb aquesta màquina virtual de Java, JVM (Java Virtual Machine). O sigui, dit d'una altra manera, el codi en Python engegarà la JVM i a partir d'aquest moment es tindrà accés al Java i al Python alhora.

Cal remarcar que el codi és en Python, o sigui el fitxer serà en l'extensió .py, i es necessitarà tenir instal·lat el Python per executar aquest codi. Si es volgués fer al revés, o sigui programar amb Java i tenir accés al Python, l'eina JPype no serviria. Però existeixen altres llibreries que disposen d'aquesta altra funcionalitat.

D'altra banda, aquest accés al Java està lligat només a una carpeta on a dins hi hagin les classes o llibreries implementades que es necessitin. Aquesta carpeta es defineix dins el codi: al arrancar el JPype s'introdueix la ruta cap a aquesta carpeta. Només es podran executar mètodes de les classes que hi hagin dins aquesta carpeta.

A continuació veurem alguns exemples per entendre el funcionament:

Primer de tot s'ha de disposar d'una classe en Java, en aquest exemple s'ha programat la classe Prova, que té una taula d'enters com a atribut, un únic constructor sense paràmetres, que instancia la taula que té d'atributs a 10 posicions. Veure Figura 7.5.1.

La classe té 3 mètodes:

- getTaula(). per retornar la taula d'atribut.
- emplenar(int x). Ficarà el valor x a totes les posicions de la taula d'atribut.
- mostrar() que escriu directament per pantalla les 10 posicions de la taula d'atribut.

És important mencionar que, per a utilitzar el JPype, cal que es defineixi un package de Java, ja que es necessitarà per a que trobi aquesta classe.

```java
package com;

public class Prova {
    private int taula[];

    public Prova() {
           taula=new int[10];
    }

    public int[] getTaula(){
      return taula;
    }

    public void emplenar(int x){
      for(int i=0; i<taula.length;i++){
      taula[i]=x;
      }
    }

    public void mostrar(){
      for(int i=0; i<taula.length;i++){
      System.out.print(taula[i]+"\n");
      }
    }
}
```

Figura 7.5.1 Classe Prova implementada en Java

En l'exemple de la Figura 7.5.1 s'instanciarà un objecte de la classe Prova cridant el seu constructor per defecte. Tot seguit es cridarà el mètode emplenar per modificar la taula que té l'objecte com a atribut, i es mostrarà per pantalla. Per mostrar la taula per pantalla es podrà fer de 2 maneres, ja que es té accés al Python i al Java alhora, o sigui es pot donar la responsabilitat al Java o al Python perquè imprimeixin. En l'exemple s'imprimirà de les 2 maneres, primer es cridarà al mètode mostrar() de la classe ja implementada, Prova, donant la responsabilitat al Java, ja que aquest mètode el que fa és imprimir per pantalla directament. Llavors, per donar la responsabilitat al Python, el que es farà serà cridar el mètode getTaula(), que en comptes d'imprimir, el que fa és retornar la taula d'enters d'atribut. El resultat s'assignarà a una nova variable en Python, per llavors fer el print de Python d'aquesta variable. A la Figura 7.5.2 es mostra el codi Python.

```
import jpype
import os.path

jpype.startJVM(jpype.getDefaultJVMPath(),
"Djava.class.path=C:/Marc/PFC/JPype/") #Obrir maquina virtual de Java

pkg = jpype.JPackage('com')  #Obtenir el package
classProva = pkg.Prova #Obtenir la classe
prova = classProva()  #crear objecte
prova.emplenar(5)     #Emplenem la taula amb 5s
prova.mostrar()       #Mostrar per pantalla la taula

taula=prova.getTaula()      #Assignem la taula a la variable taula
print taula                       #Mostrem la variable

jpype.shutdownJVM() #Tancar maquina virtual de Java
```

Figura 7.5.2 Exemple d'un codi Python utilitzant el JPype

El primer que es fa en aquest codi d'exemple, és engegar la JVM amb la comanda:

jpype.startJVM(jpype.getDefaultJVMPath(),"Djava.class.path=C:/Marc/PFC/JPype/")

Per tant es necessiten com a mínim 2 paràmetres:

- el primer és la ruta on es troba el fitxer jvm.dll. Aquest fitxer és la llibreria de la JVM, per tant ha d'estar en la carpeta on està instal·lat el Java. Per facilitar-ho el JPype ja té la comanda getDefaultJVMPath(), que troba aquest fitxer, tal com s'ha usat en l'exemple.
- I el segon paràmetre és la ruta on hi ha les classes Java que s'utilitzaran. Han d'estar ja compilades, o sigui els fitxers .class han d'estar creats. En l'exemple, dins la ruta C:/Marc/PFC/JPype/, hi haurà una carpeta “com”, el package de la classe Prova, i dins aquesta carpeta hi haurà el fitxer Prova.class.

Si fos necessari, es pot introduir un tercer paràmetre per introduir una nova ruta on hi hauria fitxers .jar.

Aleshores, seguint el codi, s'obté primer el package utilitzant la comanda JPackage() introduint com a paràmetre el package que es vol obtenir, en aquest cas el 'com'. Tot seguit ja es pot obtenir la classe que es necessiti dins el package. En aquest cas només hi ha la classe Prova. Per tant, des de la variable pkg on se li ha assignat el package 'com', obtenim la classe Prova introduint pkg.Prova. O sigui, la classe Prova ara és atribut del package.

Un cop s'ha guardat la classe Prova dins d'una variable, en l'exemple “classProva”, ja es pot accedir de manera normal a aquesta classe en Java. En l'exemple es defineix una variable prova on se li assigna la crida al constructor o sigui classProva(). Tot seguit es crida el mètode emplenar com a paràmetre el número 5. Llavors el mètode mostrar() i finalment el mètode getTaula() s'assigna a una variable nova “taula” per fer llavors el print de Python d'aquesta variable.

Per últim, a la Figura 7.5.3 veiem la sortida que ha donat el programa.

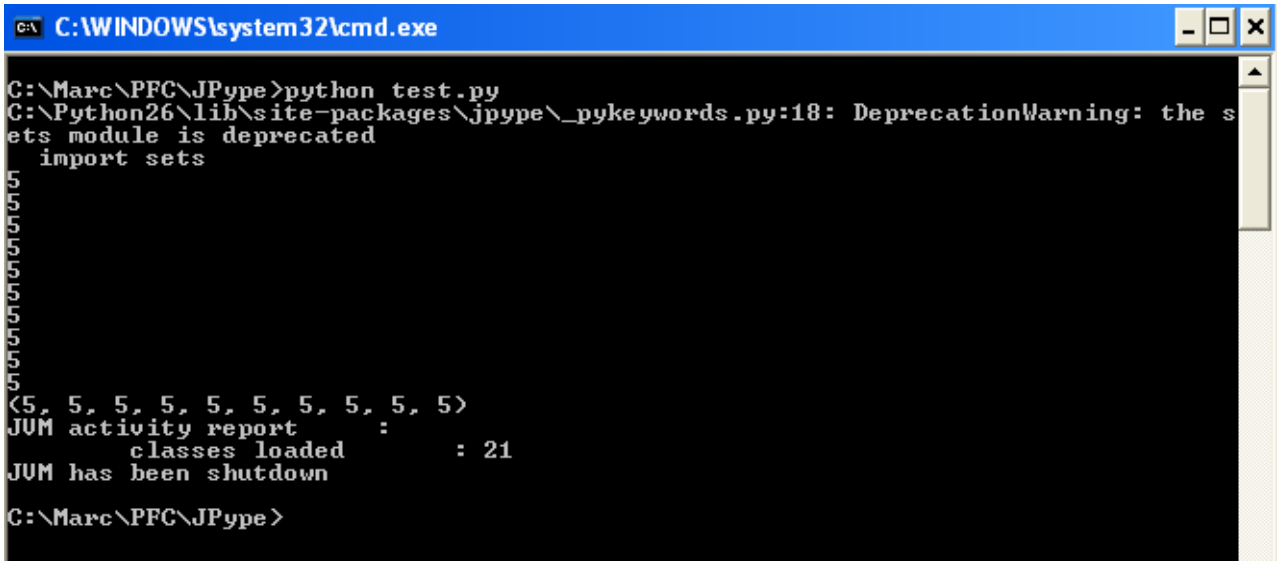


Figura 7.5.3 Resultat d'executar el codi Python anterior

Tal i com es pot veure, primer s'executa el mètode que mostra per pantalla la taula amb Java: són els 10 primers cincs amb salt de línia. Llavors es mostra la variable taula amb Python que s'havia assignat a la taula utilitzant el print propi de Python, que són els 10 cincs següents.

7.6. Subprocess

La llibreria Subprocess serveix per poder executar programes externs des de Python. Un exemple per entendre la seva utilitat seria el següent: tindria un codi Python executant-se i dins aquest codi es cridaria a un altre codi, que podria ser en qualsevol llenguatge mentre estigui tot instal·lat correctament per fer-lo funcionar al mateix ordinador, realment el que fa aquesta llibreria es com si fes una crida a la consola del sistema operatiu, (cmd en windows). Per fer-ho s'executa la següent comanda:

subprocess.Popen('comanda')

dins la cadena de caràcters 'comanda' s'introdueix el que es vol fer, tot seguit es mostra un exemple per comprendre el seu funcionament. Veure Figura 7.6.1.

Seguint l'exemple de la Figura 7.6.1 veiem que és un codi en Python. Primer de tot s'importa la llibreria subprocess, aleshores s'han indicat els “(...)” per indicar que aquest codi pot tenir qualsevol funcionalitat, i llavors mitjançant la crida específica s'indica que es vol executar el següent:

'Python C:/Marc/PFC/JPype/serverPRC.py'

Això el sistema ho interpreta com que es vol executar un codi en Python, que està ubicat a la ruta indicada.

7.7. Xmlrpclib

Aquesta llibreria serveix per realitzar una connexió client-servidor, va lligada amb la llibreria SimpleXMLRPCServer ja que aquesta última funciona com a servidor i la que s'està explicant com a client. Aquestes 2 llibreries treballen amb llenguatge Python.

Doncs la funcionalitat com a client és relativament senzilla, hi ha 2 passos: connectar-se i executar funcions.

- Connexió: Per realitzar la connexió amb el servidor s'ha d'utilitzar la següent comanda:

  variable = xmlrpclib.ServerProxy("URL:port/")

  A l'executar aquesta comanda es requereix que el servidor estigui en execució de manera que estigui escoltant el port per una possible petició d'un client.

  La variable recull tota la informació necessària de la connexió que s'estableix, i dins els paràmetres de la comanda cal introduir-hi la URL on es connecta i el port.

- Execució: Un cop s'ha establert la connexió és moment de fer funcionar el servidor, per fer-ho es necessita realitzar la següent comanda:

  resultat = variable.nomFunció(paràmetres)

  on variable significa el nom de la variable que s'ha definit al establir la connexió, i nomFunció és el nom de la funció que s'ha definit en el servidor.

Veiem un exemple a la Figura 7.7.1 per comprendre el seu funcionament.

```
import xmlrpclib

proxy = xmlrpclib.ServerProxy("http://localhost:8888/")
areaTriangle = proxy.calcularAreaTriangle(baseTriangle, alturaTriangle)
areaQuadrat = proxy.calcularAreaQuadrat(costatQuadrat)
```

Figura 7.7.1 Exemple funcionament del xmlrpclib

Seguint l'exemple mostrat en la Figura 7.7.1 veiem que primer es connecta al mateix ordinador i al port 8888, per tant s'hauria d'haver executat primer el servidor i que estigués escoltant al port 8888 perquè funcionés el codi. L'objecte que té la informació d'aquesta connexió és guardada dins la variable proxy.

Tot seguit ja es poden utilitzar les funcions que tingui definides el servidor. En aquest cas s'ha cridat a les funcions calcularAreaTriangle i calcularAreaQuadrat passant-li els paràmetres adequats. Per tant, tota la feina de càlcul la farà el servidor i un cop obtingui el

resultat l'enviarà de retorn cap a les variables que s'han assignat areaTriangle i areaQuadrat.

7.8. SimpleXMLRPCServer

Tal i com s'ha explicat en el capítol anterior, la llibreria SimpleXMLRPCServer es tracta de la implementació d'un servidor.

Per realitzar aquesta implementació calen 3 passos: crear el servidor, registrar les funcions i escoltar peticion.

- Crear servidor: Per crear el servidor només cal executar la següent comanda:

  variable = SimpleXMLRPCServer("nom", port)

  La variable recullirà la informació del servidor, i dins els paràmetres d'aquesta comanda cal introduir-hi un nom i el port on escoltarà.

- Registrar funcions: Un cop s'ha creat el servidor cal registrar les funcions que es vulgui de les que estan definides dins el codi, de manera que les que es registrin seran les que el client pugui utilitzar. Si no es registren no es podran utilitzar pel client. Doncs per registrar una funció es fa de la següent manera:

  variable.register_function(nomFunció, "nomFunciónPerCridar")

  Primer de tot es necessita la variable on es té recollida la informació del servidor ("variable" en l'exemple). Llavors com a paràmetres s'han d'introduir el nom de la funció com està definida i una cadena de caràcters per definir amb quin nom ha de cridar la funció el client (normalment es ficarà exactament el mateix nom).

- Per últim falta dir al servidor que ja pot començar a escoltar peticions de clients. Per fer-ho només cal indicar-li de la següent manera:

  variable.serve_forever()

  La variable de l'exemple té la informació del servidor quan s'ha creat. Com diu el nom "serve forever" fa que el servidor es quedi escoltant peticions "per sempre".

A continuació veiem un exemple per comprendre el seu funcionament. Veure Figura 7.8.1.

```
from SimpleXMLRPCServer import SimpleXMLRPCServer

def calcularAreaTriangle(base, altura):
    Return (base*altura)/2

def calcularAreaQuadrat(costat):
    Return costat*costat


server = SimpleXMLRPCServer("localhost",8888)
server.register_function(calcularAreaTriangle, "calcularAreaTriangle")
server.register_function(calcularAreaQuadrat, "calcularAreaQuadrat")

server.serve_forever()
```

Figura 7.8.1 Exemple de funcionament del SimpleXMLRPCServer

Seguint l'exemple de la Figura 7.8.1, primer de tot es defineixen les funcions, tot seguit es crea el servidor escoltant al port 8888 i guardant la informació a la variable server. Un cop creat el servidor es registren les 2 funcions que s'havien definit calcularAreaTriangle i calcularAreaQuadrat de manera que el client les hagi de cridar amb el mateix nom. Per últim se li indica el servidor que escolti peticions de clients.

7.9. Eclipse

Eclipse és un entorn integrat de desenvolupament de codi obert programada principalment en Java (per tant, multi-plataforma), per a desenvolupar projectes en varis llenguatges, sempre i quan s'instal·lin els connectors corresponents per a cada llenguatge de programació. L'eclipse serà útil alhora de realitzar implementacions amb Java.

7.10. Notepad++

El Notepad++ és un editor de codi font lliure que suporta diversos llenguatges de programació i funciona sota l'entorn MS Windows.

Aquest projecte, basat en el component d'edició Scintilla (un component d'edició molt potent), està escrit en C++ amb pur api win32 i STL (això assegura una velocitat d'execució ràpida i una mida del programa més petita). Està sota llicència GPL. El notepad++ serà útil per escriure codi Python.

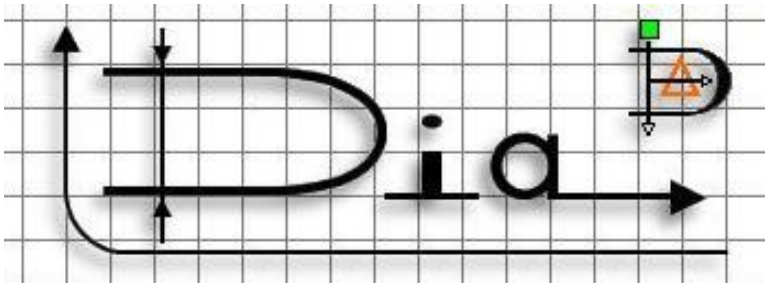
7.11. Sistema operatiu

El sistema operatiu amb el qual s'ha desenvolupat aquest projecte ha estat el Microsoft Windows XP.

7.12. Gantt project

Gantt Project és una aplicació de programari lliure que permet realitzar diagrames de Gantt. El diagrama de Gantt és una de les tècniques usades tant per l'administració pública com per l'empresa privada com a eina de gestió de la qualitat. Concretament, és una eina de planificació del treball, ja que presenta totes les activitats que s'han de realitzar i quan s'han de realitzar, i permet tenir una idea de com avança el projecte i si és necessari re-programar les actuacions planificades per tal d'adequar el projecte al nou entorn o necessitats. S'ha adaptat al treball per projectes en educació, repartint les feines del grup.

7.13. Dia

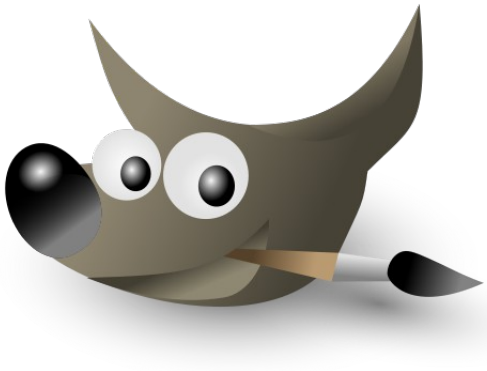


Dia és una aplicació gràfica de propòsit general per a la creació de diagrames, desenvolupada com a part del projecte GNOME. Està construïda de forma modular, amb diferents paquets de formes per a diferents necessitats.

Dia està dissenyat com un substitut de l'aplicació comercial “Visio” de Microsoft. Es pot utilitzar per dibuixar diferents tipus de diagrames. Actualment s'inclouen diagrames entitat relació, diagrames UML, diagrames de flux, diagrames de xarxes, diagrames de circuits elèctrics, etc. Noves formes poden ser fàcilment afegides, dibuixant-les amb un subconjunt d'SVG i incloent-les en un fitxer XML.

Dia s'ha utilitzat per crear qualsevol tipus de diagrama que sigui necessari per inserir dins la documentació del projecte que ajudi a entendre els conceptes.

7.14. Gimp



GIMP és un programa de GNU per al tractament d'imatges (GNU Image Manipulation Program), creat per voluntaris i distribuït sota la llicència GPL. GIMP serveix per processar gràfics i fotografies digitals. Els usos típics inclouen la creació de gràfics i logotips, el canvi de mida i retallat de fotografies, el canvi de colors, la combinació d'imatges usant capes, l'eliminació d'elements no desitjats de les imatges i la conversió entre diferents formats d'imatges. També es pot utilitzar el GIMP per crear imatges animades senzilles.

GIMP és conegut també per ser potser la primera gran aplicació lliure per a usuaris finals. Treballs anteriors, com Gcc, el nucli de Linux, etc. eren principalment eines de programadors per a programadors.

El Gimp s'ha utilitzat per manipular les figures que han estat inserides dins la documentació del projecte.

7.15. OpenOffice

OpenOffice és un projecte comunitari per crear un paquet ofimàtic basat en codi obert, amb llicència LGPL (llicència semblant a la GPL però que permet que programes propietaris usin les llibreries i aplicacions sota aquesta llicència), procedent d'una versió antiga de StarOffice de Sun Microsystems.

Pot llegir i escriure directament els fitxers creats amb els programes de Microsoft Office (Word, Excel i PowerPoint), tot i que, a diferència d'aquests, fa servir per defecte els

formats ODF.

A més està disponible per vàries plataformes, tals com Microsoft Windows, GNU/Linux, BSD, Solaris i Mac OS X. L'OpenOffice s'ha utilitzat per crear el fitxer de text i anar modificant-lo per a la realització de la documentació del treball.

8. Anàlisi i disseny del sistema

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

Abans de començar, cal dir que es va plantejar el sistema des del punt de vista de l'usuari, que per a ell fos el més senzill possible. Per tant, el primer que es va fer és dissenyar una interfície gràfica per escollir els valors dels paràmetres desitjats.

8.1. Disseny general

En aquest apartat s'expliquen les eines que es faran servir per poder desenvolupar les diferents parts del disseny d'aquest programa i es mostraran els casos que poden tenir. Per poder-ho fer millor, el primer que s'utilitzarà és un diagrama de cas d'us.

8.1.1. Diagrama de cas d'ús general

En enginyeria del programari, un cas d'ús és una tècnica per a la captura de requisits potencials d'un nou sistema o una actualització de programari. Cada cas d'ús proporciona un o més escenaris que indiquen com hauria d'interactuar el sistema amb l'usuari o amb un altre sistema per aconseguir un objectiu específic. Normalment, en els casos d'usos s'evita l'ús d'argots tècnics, preferint en el seu lloc un llenguatge més proper a l'usuari final. En altres paraules, un cas d'ús és una seqüència d'interaccions que es desenvoluparan entre un sistema i els seus actors en resposta a un esdeveniment que inicia un actor principal sobre el sistema.

Els diagrames de casos d'ús serveixen per especificar la comunicació i el comportament d'un sistema mitjançant la interacció amb els usuaris i/o altres sistemes. S'utilitzen sobretot per il·lustrar els requeriments del sistema en mostrar com reacciona a esdeveniments que es produeixen en el seu àmbit o en ell mateix.

A la Figura 8.1 es mostra el diagrama de cas d'us de context general del projecte. L'usuari té 3 accions a fer: crear planta, crear perfil i generar l'edifici.

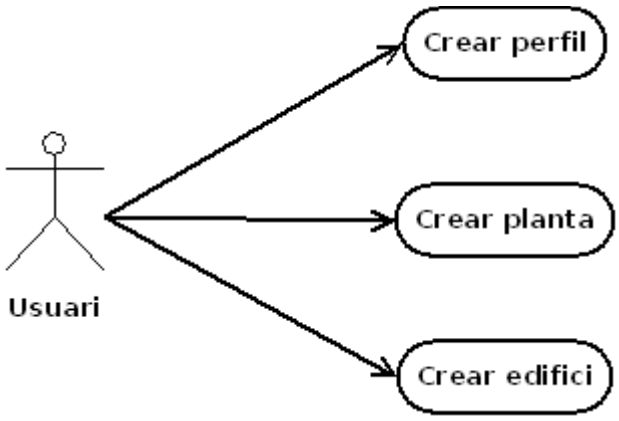


Figura 8.1 Diagrama de cas d'ús general

Fitxa	Crear planta.
Funcionalitat	Crear la geometria de la planta de l'edifici.
Actor	Usuari.
Pre-condició	Cap.
Flux principal	1. L'usuari crea el node curve del Houdini. 2. Dibuixa la forma de la planta.
Subfluxos (extensions)	Cap.
Flux alternatiu	Cap.
Post-condició	Es crea la geometria de la planta de l'edifici, (únicament en el pla XZ)

Fitxa	Crear pefil.
Funcionalitat	Crear la geometria d'un perfil de l'edifici.
Actor	Usuari.
Pre-condició	Cap.
Flux principal	1. L'usuari crea el node curve del Houdini. 2. Dibuixa la forma del perfil.
Subfluxos (extensions)	Cap.
Flux alternatiu	Cap.
Post-condició	Es crea la geometria d'un perfil de l'edifici, (únicament en el pla XY)

Fitxa	Crear edifici.
Funcionalitat	A partir d'aquest cas d'ús es crearà un edifici a partir dels paràmetres (planta i perfils) introduïts per l'usuari.
Actor	Usuari.
Pre-condició	Han d'estar definides com a mínim una planta i un perfil.
Flux principal	1. L'usuari entra els paràmetres. 1.1. Passa una geometria de la planta. 1.2. Passa una o més geometries de perfils. 2. Crea l'edifici.
subfluxos (extensions)	Cap.
Flux alternatiu	Cap.
Post-condició	S'ha creat un edifici a partir dels paràmetres (planta i perfils) donats com a entrada.

8.2. Diagrama d'activitat

El diagrama d'activitats representa els fluxos de treball pas a pas de negoci i operacionals dels components en un sistema. Un Diagrama d'Activitats mostra el flux de control general.

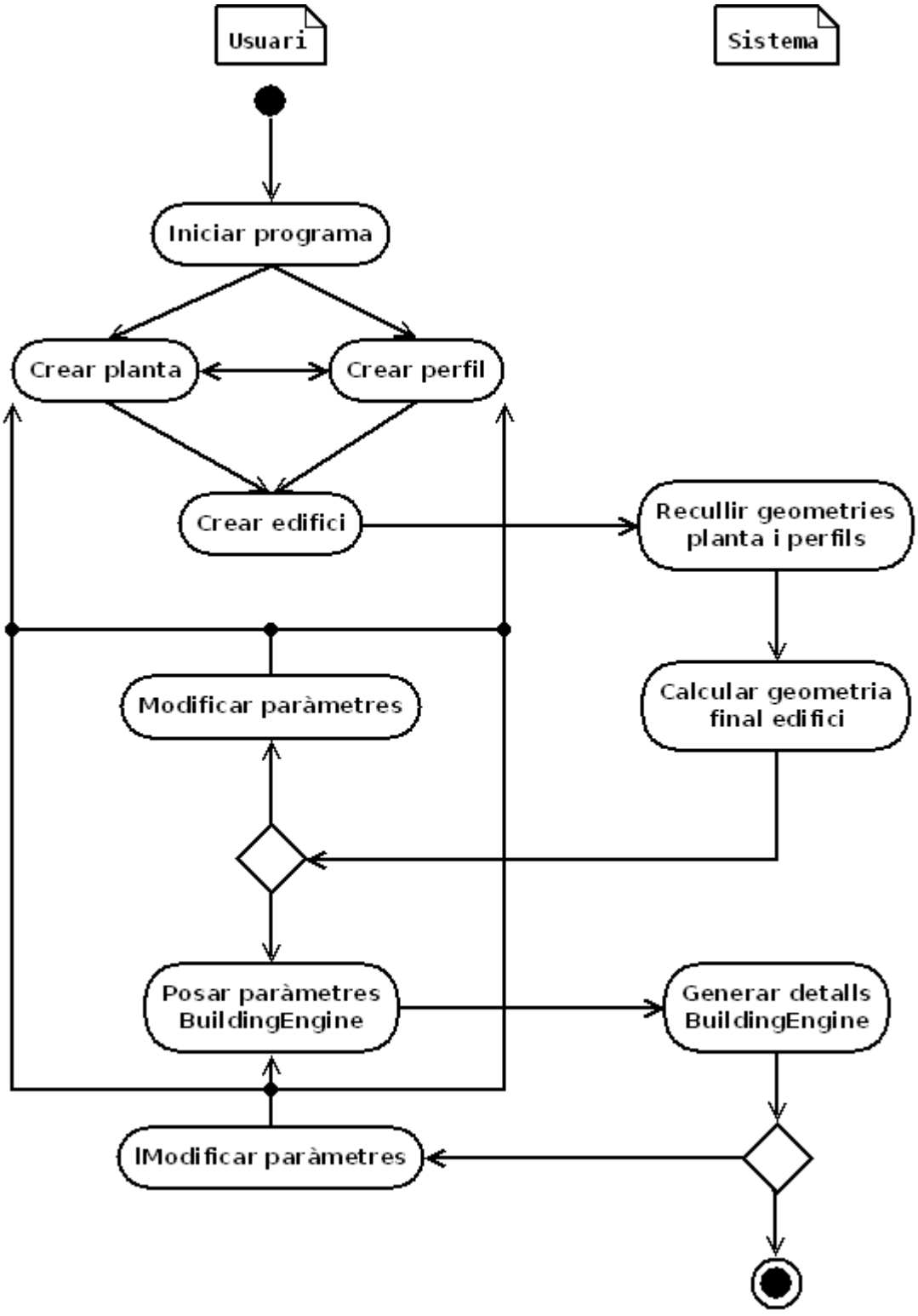


Figura 8.2 Diagrama d'activitats del sistema

El flux d'aquest projecte comença un cop l'usuari ha iniciat el programa. En aquest moment, l'usuari pot generar una planta o un perfil en l'ordre desitjat. Un cop tingui els paràmetres creats, podrà executar el procés de crear un edifici.

Quan s'executa aquest procés, el sistema primer ha de recollir les geometries d'aquests paràmetres introduïts per l'usuari, per llavors calcular la geometria que tindrà l'edifici, utilitzant el mètode d'extrusions, basades en el straight skeleton. Un cop creat l'edifici, l'usuari ja el podrà visualitzar en la interfície d'usuari, i llavors l'usuari pot decidir si anar enrere i modificar els paràmetres per canviar la forma de l'edifici, o pot seguir endavant i utilitzar les eines de les que disposa el buildingEngine.

Després d'inserir algun accessori per l'edifici amb buildingEngine, l'usuari pot tornar a decidir si vol anar enrere fins a tornar a modificar els paràmetres inicials, planta i perfils, per editar la forma de l'edifici, o editar els paràmetres del buildingEngine.

8.3. Diagrama de classes

Un diagrama de classes és un tipus de diagrama estàtic que descriu l'estructura d'un sistema mostrant les seves classes, atributs i relacions entre ells. Els diagrames de classes són utilitzats durant el procés d'anàlisi i disseny dels sistemes, on es crea el disseny conceptual de la informació que tindrà el sistema, i els components que s'encarreguen del funcionament i de la relació entre un i l'altre.

Tot seguit es presentaran totes les classes que s'utilitzen dins d'aquest projecte. Aquestes classes les podem separar en 2 grups, les que s'utilitzen que ja són existents i les que s'han creat per completar el projecte:

Les classes existents:

- Houdini
- Llibreria buildingEngine
- Llibreria JPype
- Llibreria CampSkeleton
- Llibreria subprocess
- Llibreria xmlrpclib
- Llibreria SimpleXMLRPCServer

I les classes creades:

- Procedural Extrusions
- JPypeServer
- serverPRC
- Wrapper

A la Figura 8.3 es presenta el diagrama de classes per veure com es relacionen entre elles, les classes de color vermell seran les ja existents, i de color verd les que s'han creat:

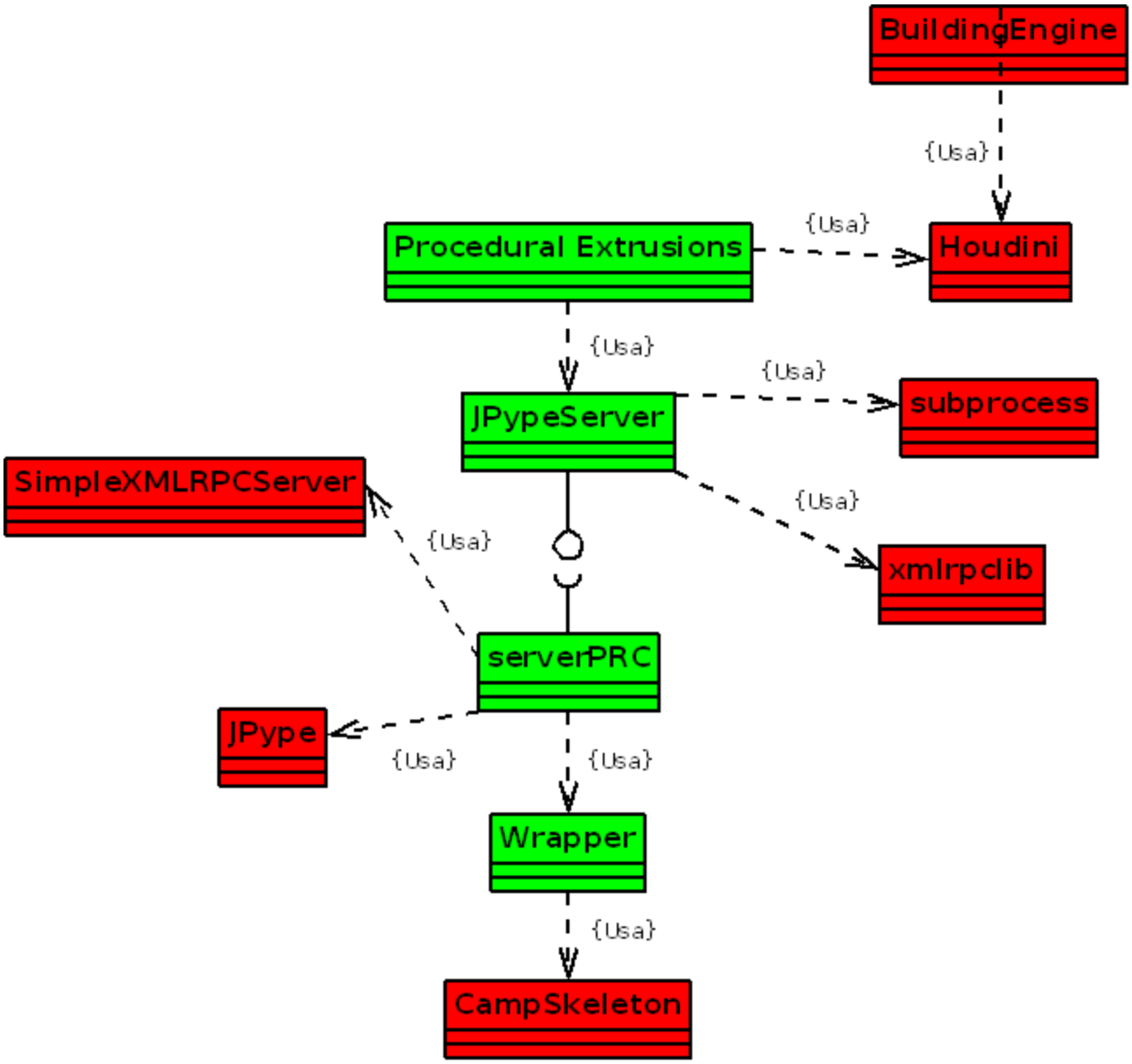


Figura 8.3 Diagrama de classes del sistema

Primer de tot s'explicarà per sobre el disseny general de les classes que hi ha dins d'aquest projecte, per llavors poder aprofundir dins de cada classe. El projecte es pot dividir en 3 parts diferenciades:

- Primer de tot, hi ha la implementació en Python per tal que quedi integrada amb el Houdini. Aquesta implementació s'encarregarà de recollir les geometries de la planta i perfils que hagi introduït l'usuari per tal d'enviar-les com a paràmetres al codi original, CampSkeleton. Aquest codi té implementada la tècnica de Straight Skeleton, perquè pugui calcular la geometria de l'edifici adequant-se als paràmetres que vol l'usuari.

  L'altra tasca que ha de fer és recollir la geometria de l'edifici resultant, que vindrà donada pel codi original, com s'ha dit abans.

- Una altra part del projecte és el codi CampSkeleton que té implementada la tècnica de Straight Skeleton, en Java. Per tant aquest codi el que farà serà rebre com a paràmetres la planta i perfils introduïts per l'usuari, i retornar la geometria de l'edifici complet.
- Com que el codi CampSkeleton no s'ha modificat i està en Java, es necessita una nova classe que funcioni per enllaçar les 2 parts anteriors, Python amb Java. El primer que farà serà rebre els paràmetres de planta i perfils des del Python. Per tant s'hauran de convertir en Java, i a més, l'estructura de dades d'aquestes geometries també s'hauran de modificar, de manera que siguin compatibles amb el codi original.

  Un cop aconseguit tot això, aquesta classe farà crides al mòdul CampSkeleton per tal que només s'utilitzi la part que interessa pel projecte.[1]

  Tot seguit recollirà la geometria de l'edifici resultant, per convertir l'estructura de dades compatible amb la del Houdini.

A la Figura 8.4 es mostra un esquema del que s'acaba d'explicar.

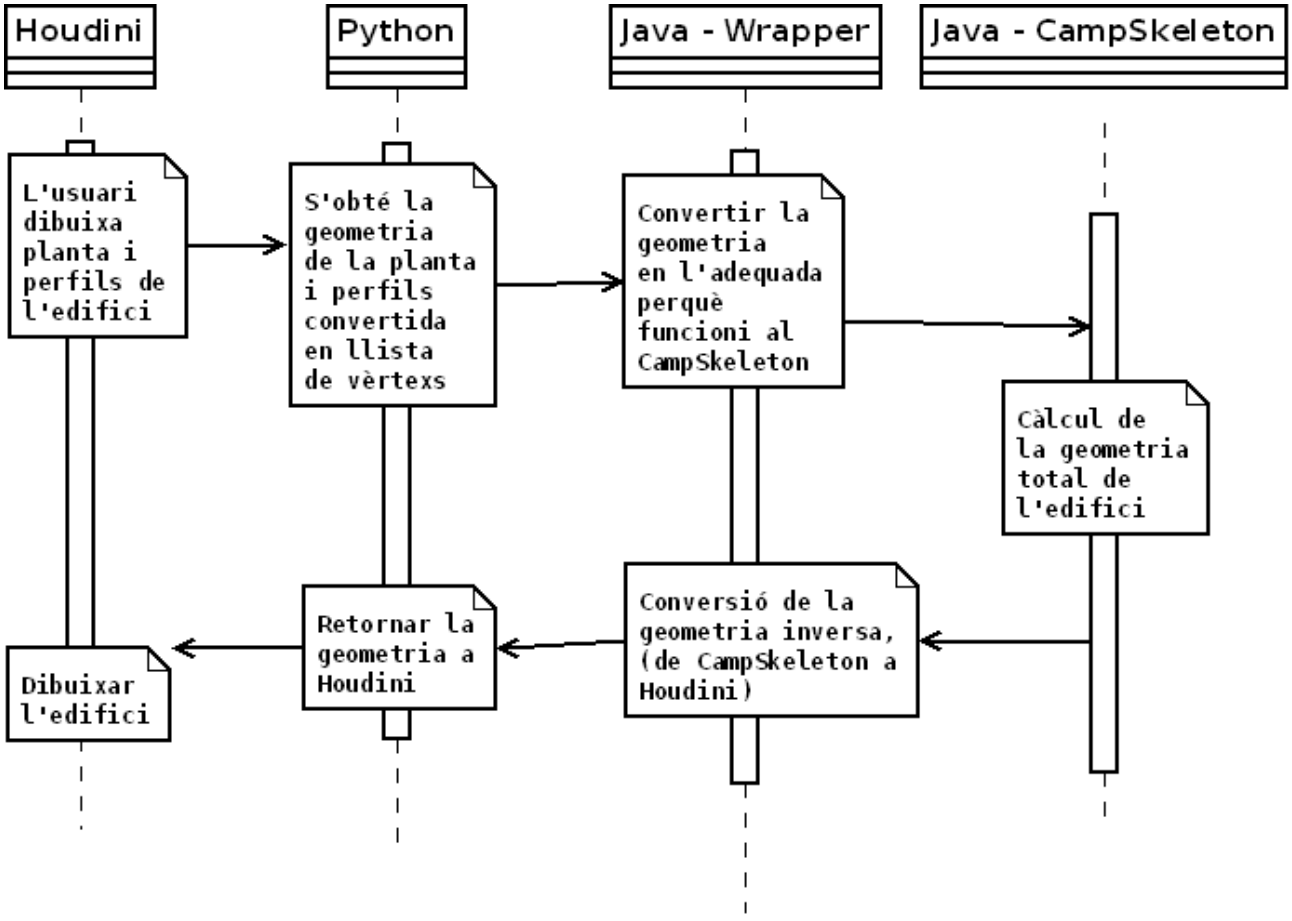


Figura 8.4 Esquema del disseny general del sistema

1 Recordar que aquest codi complet és el SitePlan que té 2 parts, una part és la tècnica de Straight Skeleton i l'altre són accessoris per adornar l'edifici. També disposa d'una interfície d'usuari. Per tant només fa falta la implementació de l'Straight Skeleton, o sigui la classe CampSkeleton.

A continuació s'explicarà en detall la funcionalitat que té cada classe i com interactuen entre elles.

8.4. ProceduralExtrusions

Aquesta és la classe principal. El flux del sistema s'origina en aquesta classe i fa que s'enllaci amb les altres, i els resultats acaben retornant també a aquest lloc. El codi d'aquesta classe té les responsabilitats següents:

- Crear objecte JPypeServer i engegar-lo.
- Recollir les geometries de Houdini, planta i perfils.
- Convertir les geometries en llistes de vèrtexs.
- Deixar la responsabilitat de crear l'edifici a la classe JPypeServer i per tant recollir la geometria de l'edifici resultant (ja compatible amb Houdini)
- Dibuixar l'edifici.
- Tancar el servidor.

A continuació es mostra el diagrama UML de la classe ProceduralExtrusions.

ProceduralExtrusions
getPlanta(self) getPerfils(self) convertPlanta(self,geometriaPlanta) convertPerfils(self, geometriaPerfils) convertBuilding(self,estructures,geometria) main(self)

- **getPlanta i getPerfil:** Aquests mètodes s'encarreguen de recollir la geometria de Houdini, la planta i els perfils que han estat introduïts com a paràmetres.
- **convertPlanta i convertPerfils:** Tenen com a paràmetre d'entrada la geometria de Houdini de la planta o els perfils. El seu objectiu és convertir aquestes geometries en llistes de vèrtexs per poder ser utilitzades més tard.

  La planta i el perfil tindran la mateixa estructura de dades: una llista de vectors. En el cas de la planta serà bidimensional ja que només importarà el pla XZ, i si hi hagués component Y en algun vector, es suprimiria deixant-lo a 0. En canvi en els perfils encara que no importi la component Z perquè el perfil està en el pla XY, sí

que importa la situació de on es troba el perfil, per poder associar els costats de l'edifici als perfils. Per tant l'estructura de dades dels perfils serà una llista de vectors tridimensionals. Cal recordar que la component Z ha de ser igual en tots els vèrtexs del perfil. En cas que sigui diferent, el programa es quedaria amb la forma del perfil formada en el pla XY, i la situació del perfil la recolliria en el primer vèrtexs que defineix el perfil.

Exemple d'una planta quadrada:

X	Z
5.0	5.0
5.0	-5.0
-5.0	-5.0
-5.0	5.0

I un de perfil:

X	Y	Z
0.0	0.0	0.0
0.0	2.0	0.0
4.0	5.0	0.0

Cal dir que la planta només pot ser una llista de vectors bidimensionals, però en el cas dels perfils, com que l'usuari en pot definir més d'un, l'estructura es convertiria en una llista de llistes. Per exemple:

Núm perfil	X	Y	Z
0	0.0	0.0	3.0
	0.0	2.0	3.0
	4.0	5.0	3.0
1	0.0	0.0	7.0
	0.0	4.0	7.0
	3.0	7.0	7.0

- **convertBuilding:** És el mètode encarregat de dibuixar l'edifici, té com a paràmetres d'entrada elcontenidor de Houdini, (on s'ha de dibuixar) i la variable

estructures formada per 2 llistes, una dels vèrtexs i l'altre de les cares que té l'edifici, de manera que ja es pot dibuixar l'edifici. A la Figura 8.4.1 es mostra el pseudocodi.

```
convertBuilding(estructures, geo):

        Vertexs = estructures[0]
        Cares = estructures[1]

        Per c en Cares Fer
        Poligon = crearPoligon()

              Per cada vertex en c Fer
              V = crearVertex(vertex)
              Poligon.afegirVertex(v)

        Fi Per
   Fi Per
```

Figura 8.4.1 Pseudocodi del mètode convertBuilding

- **main:** És el mètode principal, s'encarrega de coordinar-ho tot.

  Abans de fer res, s'ha d'encarregar de crear i engegar el servidor perquè més tard estigui disponible. Per fer-ho, com s'ha dit anteriorment, només cal crear l'objecte de la classe JPypeServer i cridar a la funció d'aquesta classe “engegarServer()”.

  Tot seguit, s'ha de recollir la geometria de Houdini. Per fer-ho es criden els mètodes getPlanta i getPerfils.

  Llavors es necessita convertir aquestes geometries en estructures de dades adequades per utilitzar-les posteriorment, de manera que es criden els mètodes convertPlanta i convertPerfil per a que retornin una llista de vèrtexs que representa la planta, i una altra que representi el perfil (llista de llista de vèrtexs en el cas que hi hagi més d'un perfil).

  Un cop emplenades aquestes variables es deixa la responsabilitat de crear l'edifici a la classe JPypeServer cridant a la funció “callServer”, passant-li els paràmetres planta i perfils que s'acaben de crear. Aquesta funció retornarà la geometria de l'edifici resultant, ja amb l'estructura de dades adequada per Houdini. Aquesta estructura de dades és la següent:

  - Una llista de vectors tridimensionals representant els vèrtexs de l'edifici en l'espai 3D.
  - Una llista on cada posició representarà quins vèrtexs estan formant cada cara de l'edifici, per tant cada posició serà variable depenent de si s'han format cares triangulars, quadrades, etc.

  Un cop es disposi d'aquestes llistes, ja serà molt senzill dibuixar-ho utilitzant les funcions específiques del Houdini. Per fer-ho es crida la funció convertBuilding passant com a paràmetres una variable que tingui encapsulada a dins les 2 llistes que s'han obtingut anteriorment (vèrtexs i cares)

A continuació es mostra el pseudocodi de la funció main ja que és la encarregada de fer-ho tot.

```
Main():
   j = Crear JpypeServer()
   s = j.engegarServer()

   geometria = Houdini.ObtenirGeometria()
   planta = convertPlanta(getPlanta())
   perfils = convertPerfils(getPerfils())

   estructures = j.callServer(planta,perfils)

   convertBuilding(estructures, geometria)
   j.tancarServer(s)
```

Figura 8.4.2 Pseudocodi de la funció main de ProceduralExtrusions

8.5. JPypeServer

La funcionalitat d'aquesta classe és molt senzilla, s'ha d'encarregar d'engegar el servidor, per més tard fer una crida a aquest. Així doncs el diagrama UML d'aquesta classe és el següent:

JPypeServer
engegarServer(self) callServer(self,planta,perfils) tancarServer(self,s)

Com s'ha vist en l'explicació de la classe anterior ProceduralExtrusions, la classe JPypeServer l'únic que fa es executar les seves funcions quan la classe ProceduralExtrusions ho demana. Així doncs l'ordre en què es criden aquestes funcions és el mateix, tal i com està ordenada al diagrama UML de la classe. A continuació s'explica què fa cada funció:

- **engegarServer:** Utilitza la llibreria subprocess, tal i com s'ha explicat en el tema 7.6, per poder engegar el servidor. Aquest servidor es tracta d'un codi en Python independent, que s'explicarà més endavant. Veure secció 8.6. Gràcies al subprocess, es pot executar de manera externa, fent que el servidor quedi executat de forma independent.

- **callServer:** Utilitza la llibreria xmlrpclib, tal i com s'ha explicat en el tema 7.7, per poder fer de client del servidor que s'ha executat anteriorment. L'objectiu d'aquest mètode és executar una funció del servidor passant-li com a paràmetres la planta i els perfils, ja que se li dóna la responsabilitat de crear l'edifici en aquest servidor. Per tant, un cop executada aquesta funció, ja es rep l'estructura de dades adequada de l'edifici resultant. El pseudocodi d'aquest mètode es mostra a la Figura 8.5.1.

```
callServer(self,planta,perfils):
   Import xmlrpclib
   Proxy = xmlrpclib.ServerProxy("localhost:8888")
   Estructures = proxy.main(planta,perfils)
   Return estructures
```

Figura 8.5.1 Pseudocodi del mètode callServer

- **tancarServer:** el paràmetre s té la informació del servidor. Per tant l'únic que s'ha de fer és tancar el servidor.

8.6. ServerPRC

Primer de tot, aclarir que aquesta classe s'ha hagut d'implementar perquè el Houdini donava problemes si s'utilitzava la llibreria JPype directament a la classe ProceduralExtrusions. Això possiblement és degut a que, com que aquesta classe s'executa directament al Houdini, s'utilitza un Python lleugerament diferent al que s'instal·la al sistema operatiu.

Per tant, vist que el JPype funcionava correctament amb el Python instal·lat al sistema operatiu, es va decidir crear una nova classe en Python, fora del Houdini.

Dit això, seguint el que s'ha explicat en la classe JPypeServer, primer de tot cal explicar que, per a què aquesta classe treballi com a servidor, s'ha utilitzat la llibreria SimpleXMLRPCServer, tal i com s'ha explicat en el tema 7.8.

Recordant el que s'ha vist a la secció anterior, 8.5, el JPypeServer a cridat a la funció main de la classe ServerPRC amb 2 paràmetres, planta i perfils, de manera que se li havia deixat la responsabilitat de crear l'edifici a la classe ServerPRC.

Llavors, la funcionalitat d'aquesta classe és utilitzar el JPype per interactuar amb la classe Wrapper, que està implementada amb Java.

El diagrama UML d'aquesta classe és molt simple, ja que només es necessita el mètode main.

ServerPRC
main(planta,perfils)

El mètode principal d'aquesta classe, main, es mostra a continuació. Veure figura 8.6.1.

```
main(planta,perfils):
   engegarJpype()
   wrapper = Crear Wrapper(planta,perfils)
   wrapper.emplenarEstructura()

   vertexs = wrapper.getVertexs()
   cares = wrapper.getCares()

   estructures = [vèrtexs,cares]

   tancarJpype()
   return estructures
```

Figura 8.6.1 Pseudocodi de la funció main de ServerPRC

Observant el pseudocodi de la figura 8.6.1, primer de tot s'engega el JPype tal i com s'ha explicat en el tema 7.5. Un cop engegat, ja podem executar funcions implementades en Java. Primer creem l'objecte Wrapper utilitzant el constructor per paràmetres, i es passa la planta i perfils. Aquest constructor té la responsabilitat de crear l'edifici.

Per tant, un cop creat l'objecte Wrapper, ja es té l'edifici resultant calculat, però l'estructura de dades d'aquest edifici és la de CampSkeleton, i no és compatible amb Houdini. Per tant es necessita fer una conversió. Per fer-ho s'utilitza la funció emplenarEstructura de la classe Wrapper, que emplenarà les variables que interessen: les 2 llistes de vèrtexs i cares.

Dit això, només fa falta cridar als mètodes getVertexs() i getCares() per poder recollir aquestes estructures.

Per acabar només fa falta encapsular les 2 llistes en una variable, tancar el JPype i retornar la variable que encapsula les llistes.

8.7. Wrapper

El Wrapper és una classe que ajuda a utilitzar la llibreria CampSkeleton, per tal d'executar els mètodes necessaris. Concretant el què s'ha explicat anteriorment, la classe Wrapper té la responsabilitat de les següents tasques:

- Convertir l'estructura de dades de Houdini a CampSkeleton.
- Crear edifici.
- Convertir l'estructura de dades de CampSkeleton a Houdini.
- Retornar l'estructura de dades.

A continuació es mostra el diagrama UML de la classe Wrapper.

Wrapper
vertexs: Taula de floats cares: Taula d'enters s: Skeleton (És l'esquelet de l'edifici)
Wrapper(planta,perfils) (constructor) createPlan(Plan,planta,perfils) makeProfiles(perfils) getClosestProfile(Point2d p1, Point2d p2,perfils,allProfiles){ emplenarEstructura() getVertexs() getCares() trobarVertex(taula) existeix(taula)

Encara que anteriorment s'hagi dividit tota la feina que fa el Wrapper en 4 tasques, en general es pot dividir únicament en 2 parts, crear l'edifici i conversió variables. Per a crear l'edifici s'utilitzen les funcions: constructor, createPlan, makeProfiles, i getClosestProfile. Les altres serveixen per fer la reconversió de l'estructura de dades en l'adequada pel Houdini, i per retornar aquestes variables.

Dit això, primer s'explicaran els mètodes encarregats de crear l'edifici i llavors els de la reconversió.

- **makeProfiles:** Aquest mètode serveix per convertir tots els perfils canviant la seva estructura de dades per l'adequada de CampSkeleton, de manera que rep com a paràmetres els perfils sense convertir, i els retorna convertits.

- **getClosestProfile:** La funcionalitat d'aquest mètode és trobar quin perfil està més a prop d'una determinada façana de l'edifici, de manera que, com a paràmetres d'entrada s'han d'introduir els 2 punts que formen la façana, la llista de perfils sense convertir, i convertits.

- **createPlan:** L'objectiu d'aquest mètode és encapsular tota la informació prèvia necessària dins l'objecte Plan per a crear l'edifici a posteriori. A continuació es mostra el pseudocodi, veure Figura 8.7.1.

```
createPlan(plan, planta, perfils):
   allProfiles = makeProfiles(perfils)

   Per cada vertex de planta Fer:
        p1 = Crear Punt2D (vertex[0], vertex[1])
        p2 = Crear Punt2D (vertex + 1[0], vertex + 1 [1])
        b = Crear Bar(p1,p2)

        profile = getClosestProfile(p1,p2,perfils,allprofiles)
        plan.profiles.put(b,profile)
```

Figura 8.7.1 Pseudocodi del mètode createPlan

Seguint el pseudocodi de la Figura 8.7.1, veiem que primer es crida al mètode makeProfiles per convertir tots els perfils canviant la seva estructura de dades per l'adequada de CampSkeleton, i es guarda a la variable allProfiles.

Llavors s'entra en un bucle que s'encarregarà d'anar creant tots els costats de l'edifici utilitzant l'objecte Bar del mòdul de CampSkeleton, per llavors associar en aquest "bar" el perfil que estigui més aprop. Per fer-ho s'utilitza el mètode getClosestProfile passant, els 2 punts en 2D que formen el costat, la llista que conté tots profiles ja convertits, i els perfils que s'havien passat des de Houdini.

Un cop s'associa un perfil a un costat, ja es pot guardar dins la variable plan. D'aquesta manera després d'haver fet tot el recorregut, l'objecte plan ja tindrà tota la informació guardada.

- **Wrapper:** El constructor del Wrapper s'encarrega de coordinar-ho tot, després d'haver-lo executat ja es tindrà l'edifici creat. A continuació es mostra el pseudocodi del constructor. Veure Figura 8.7.2.

```
Wrapper(planta,perfils):
   Plan plan = Crear Plan()
   createPlan(plan,planta,perfils)
   S = Crear PlanSkeleton(plan)
   s.Skeleton()
```

Figura 8.7.2 Pseudocodi del constructor del Wrapper

Seguint el pseudocodi de la Figura 8.7.2, primer de tot es crea l'objecte Plan del mòdul de CampSkeleton, utilitzant el seu constructor per defecte. Llavors es crida al mètode createPlan, per unir tota la informació necessària per a crear l'edifici.

Un cop es tingui tota aquesta informació encapsulada en l'objecte Plan, es crea l'objecte PlanSkeleton utilitzant el constructor per paràmetres i passant-li el Plan. D'aquesta manera ja s'estarà totalment preparat per a crear l'edifici.

L'únic que falta és cridar el mètode Skeleton() de la classe PlanSkeleton que crearà l'edifici resultant. Un cop executat el constructor, l'edifici resultant quedarà guardat dins l'atribut de la classe “s”.

- **getVertexs i getCares:** Aquests mètodes serveixen per retornar les llistes de vèrtexs i cares que estan guardades com a atributs de la classe.

- **existeix i trobarVèrtex:** Aquests mètodes seran útils per quan s'estigui reconvertint l'estructura de dades de CampSkeleton a les llistes de vèrtexs i cares. Reben com a paràmetres d'entrada un vector de 3 dimensions, de manera que el mètode **existeix** retornarà cert si aquest vector ja existeix en l'atribut de la llista de vèrtexs. En canvi el **trobarVertex** retornarà la posició de la llista de vèrtexs on està ubicat el vector que s'ha passat com a paràmetres.[2]

- **emplenarEstructura:** És el mètode encarregat de coordinar tota la reconversió de l'estructura de dades de l'edifici que ja està creat. El seu objectiu és recollir la geometria de l'edifici amb l'estructura pròpia del CampSkeleton, i convertir-la en el format que ens interessa, una llista de vèrtexs i una altra de cares. A continuació es mostra el pseudocodi. Veure Figura 8.7.3.

2 Si es crida el mètode trobarVertex se suposa que el vector que s'ha passat com a paràmetres ja hi és a la llista de vèrtexs.

```
EmplenarEstructura():
   int nVertexs=0
   int nCares=0

   Per cada cara en s.Cares() Fer:
      int vertexsPerCara=0

      Per cada vertex en cara Fer:

         vector = cara.obtenirComponents()

         Si (No existeix(vector, vertexs) Fer:

            vertexs.afegir(vector)
            cares[nCares][vertexsPerCara] = nVertexs
            nVertexs++

         Altrament

            cares[nCares][vertexsPerCara]=trobarVertex(vector,vertexs)

         FI Si

         VertexsPerCara++

      Fi Per

      Ncares++

   Fi Per
```

Figura 8.7.3 Pseudocodi de la funció emplenarEstructures

La idea general, que es pot apreciar en el pseudocodi de la Figura 8.7.3, és que es vol emplenar 2 llistes, una de vèrtexs i una de cares, on la llista de vèrtexs haurà de contenir tots els vèrtexs de l'edifici, però sense tenir-ne de repetits, i la de cares haurà de contenir totes les cares que té l'edifici. Per indicar quins vèrtexs formen cada cara es farà mitjançant la llista creada anteriorment.

Per crear un algorisme que faci això, s'han necessitat 3 variables enteres: **nVertexs** que representarà el total de vèrtexs que hi ha en cada moment mentre s'executa l'algorisme, sense comptar els repetits. **nCares** que representarà el total de cares que té l'edifici en cada moment mentre s'executa l'algorisme. Per últim **vertexsPerCara** representa el número de vèrtexs que té la cara actual, ja que, com s'ha explicat anteriorment, cada cara pot tenir un número de vèrtexs diferent.

Primer de tot s'inicialitzen les variables **nVertexs** i **nCares** a 0, ja que les estructures estan buides. Llavors s'ha de fer un recorregut per a cada cara que té l'edifici. Per fer-ho l'objecte PlanSkeleton ja té un mètode. D'aquesta manera, un cop a dins del “Per”, s'inicialitza la variable **vertexsPerCara** a 0, ja que en aquest moment estem accedint a una cara però encara no s'ha obtingut cap vèrtex. El següent pas és crear un altre bucle

per accedir a tots els vèrtexs que compon la cara. Dins d'aquest bucle inicialitzem una variable auxiliar vector on es guarden les components del vèrtex al que s'ha accedit. En aquest moment s'ha de mirar si aquest vèrtex ja s'ha introduït anteriorment a la llista de vèrtexs. Per fer-ho, s'ha implementat la funció existeix() on se li passa per paràmetres la llista i el vector que es vol mirar si ja està guardat, de manera que retornarà cert si existeix i fals altrament.

A continuació, es pregunta si aquest vector existeix. Si la funció retorna fals, s'afegeix el vector dins la llista de vèrtexs, i a més s'assigna a la llista de cares.

Si al preguntar si el vèrtex existeix es dóna el cas afirmatiu, ja no cal afegir-lo a la llista de vèrtex. El procediment és buscar a quina posició està el vèrtex, utilitzant la funció implementada trobarVertex(), on se li passa per paràmetres el vector i la llista de vèrtexs, i retorna un enter indicant la posició on es troba. Així doncs, no cal afegir el vèrtex perquè ja existeix, però sí que cal afegir-lo a la llista de cares, fent servir l'enter que acaba de retornar la funció trobarVertex().

Un cop s'ha executat tota la funció d'emplenarEstructures(), ja es tenen les llistes de vèrtexs i cares emplenades, i assignades com a atributs de l'objecte Wrapper. Per tant, per retornar-les cap al serverPRC, només cal que es cridin els mètodes getVertexs i getCares.

A continuació es mostrarà un diagrama per observar les interaccions que hi ha entre les classes que s'han explicat en aquest capítol. Veure Figura 8.7.4.

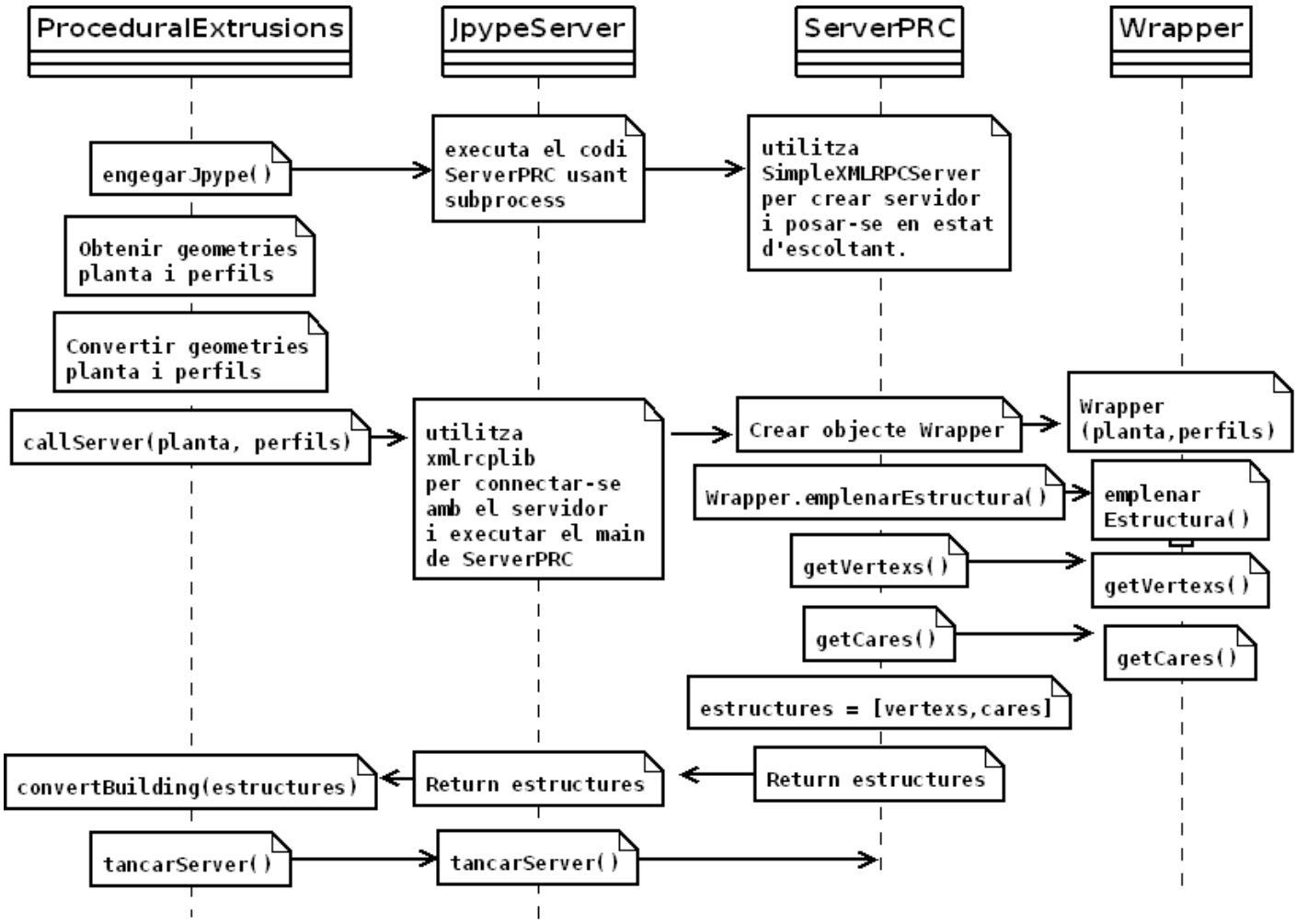


Figura 8.7.4 Interactivitat entre les classes

Seguint el diagrama de la figura 8.7.4, es veu que el que té el control total és la classe ProceduralExtrusions. Primer de tot engega el servidor ServerPRC, per fer-ho crida a una funció de la classe JPypeServer que aquesta fa engegar la classe ServerPRC. Doncs a partir d'aquí, la classe ServerPRC es va executant de manera que quedi tot preparat per rebre peticions. Mentre el ServerPRC es va executant, el ProceduralExtrusions va continuant la seva execució, obté les geometries de la planta i perfils que hagi introduït l'usuari, i les converteix amb llistes de vèrtexs. A partir d'aquí crida a la funció callServer de JPypeServer passant-li la planta i els perfils. El JPypeServer un cop li han cridat que executi aquesta funció, realitza una petició al servidor ServerPRC per cridar una funció d'aquesta classe, passant-li els paràmetres que li havien passat, planta i perfils. El ServerPRC executa la funció que ha demanat el JPypeServer i fa el següent: crea l'objecte Wrapper i l'inicialitza cridant el seu constructor passant-li la planta i els perfils, de manera que un cop executat aquest constructor, l'objecte Wrapper ja té dins el resultat de l'edifici. Per reconvertir l'estructura de l'edifici en l'adequada crida al mètode del Wrapper emplenarEstructura. Llavors obté les llistes necessàries, vèrtexs i cares, cridant a les funcions getCares i getVertexs, per llavors encapsular-les dins una nova variable estructures, i retornar-ho. Un cop s'ha retornat a la classe JPypeServer, aquest també ho retorna cap a ProceduralExtrusions, que per finalitzar, crida a la funció convertBuilding per dibuixar l'edifici i tenca el servidor.

9. Implementació i proves

Aquest capítol tracta sobre els passos que s'han seguit a l'hora de realitzar aquest projecte. Per entendre la implementació, cal parlar de les utilitats que s'utilitzen del Houdini i de les implementacions de totes les classes, mostrant-ne exemples.

9.1. Creació d'un nou node de Houdini

Per poder utilitzar el codi Python s'ha de crear un node programable de Houdini. Per crear-ne un s'ha d'anar al menú File i anar a la a la opció de New Operator Type. Veure Figura 9.1.

En aquesta finestra que s'ha obert, permet escollir el nom del node, l'etiqueta del node, l'estil, el tipus i on desar el node en format de llibreria de nodes (.otl). Una vegada creat, ja es podran configurar els paràmetres que rebrà el node.

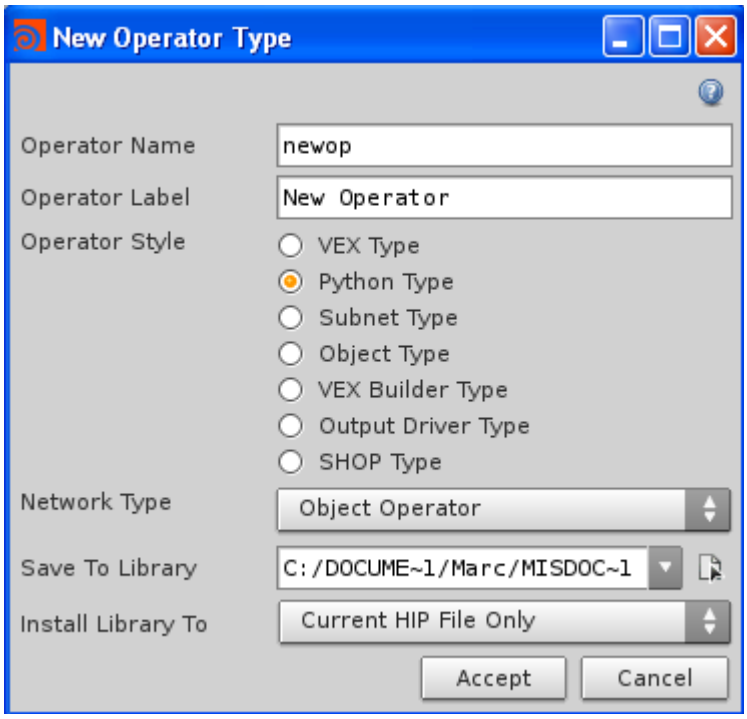


Figura 9.1 Interfície per crear un nou node

9.2. GUI i PythonSOP

Un cop creat el nou node, Houdini mostra una interfície gràfica per l'usuari, on podrà editar aquest nou node perquè realitzi les tasques objectives.

Disposa de moltes pestanyes, opcions bàsiques, paràmetres, ajuda, codi, etc.

Pels objectius del projecte només ha estat necessari modificar en opcions bàsiques: assignar-li en aquest node un mínim i màxim d'entrades igual a 2, una entrada rebrà la planta i l'altre el o els perfils.

I d'altra banda modificar l'apartat de codi, que és el que executarà aquest node.

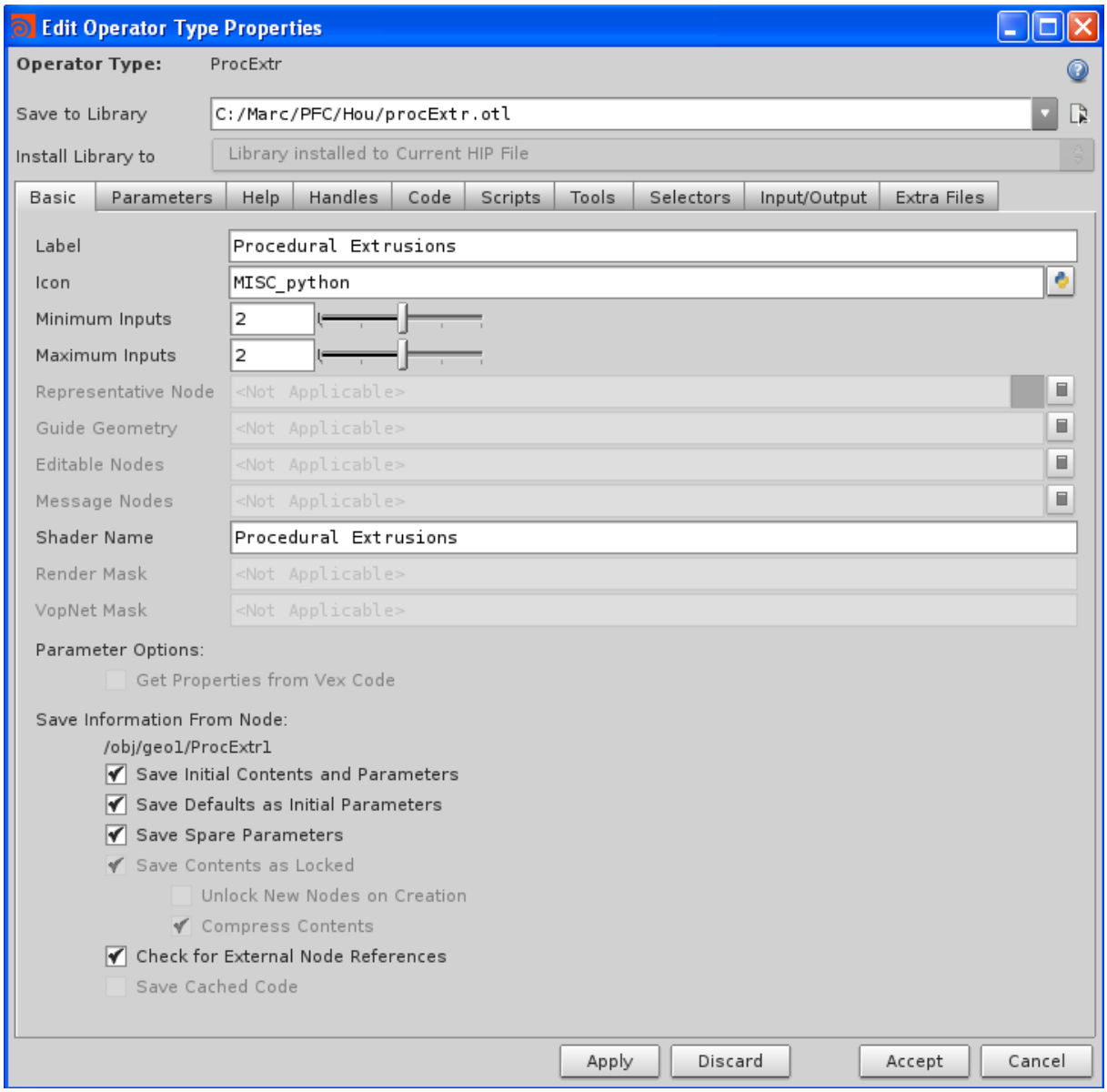


Figura 9.2 Interfície gràfica d'usuari de Houdini (bàsic)

En la Figura 9.2 es mostra la interfície gràfica d'usuari de Houdini, dins aquesta finestra hi

ha totes les possibles opcions per editar el node.

A la pestanya actual “bàsic” s'hi assignen les opcions més bàsiques, per al programa desenvolupat al llarg d'aquest projecte ha calgut editar el mínim i màxim número d'entrades a 2, ja que en una serà l'entrada de la planta i l'altre el conjunt de perfils.

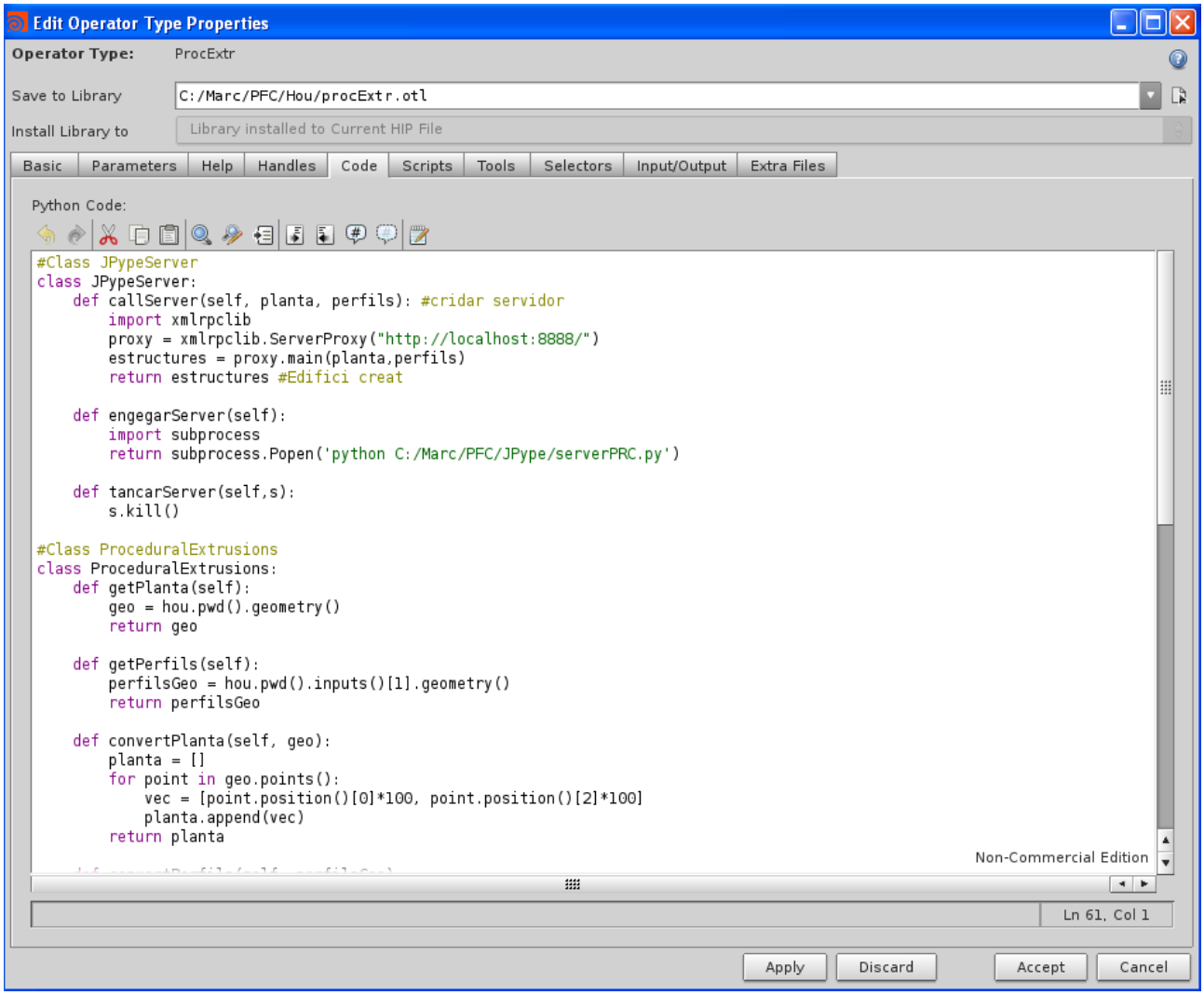


Figura 9.3 Interfície gràfica d'usuari de Houdini (codi)

En la Figura 9.3 es mostra l'altre i última modificació del node que s'ha fet, doncs ha sigut introduir tot el codi necessari que farà possible calcular la geometria de l'edifici.

Per concretar, dins aquest node existeixen les classes ProceduralExtrusions i JPypeServer que s'han explicat en el capítol 8, anàlisi i disseny del sistema.

9.3. HDA Procedural Extrusions

Resulta que introduint els paràmetres de la planta en el pla XZ i perfils en XY, donava el resultat de l'edifici orientat en l'eix Z com a normal de l'edifici, o sigui 90º rotat amb l'eix X,

per tant s'ha decidit fer un nou node HDA que encapsuli el node ProceduralExtrusions i hi apliqui una rotació de 90º al eix X perquè quedi l'edifici correctament orientat. A la Figura 9.4 es mostra l'encapsulació.

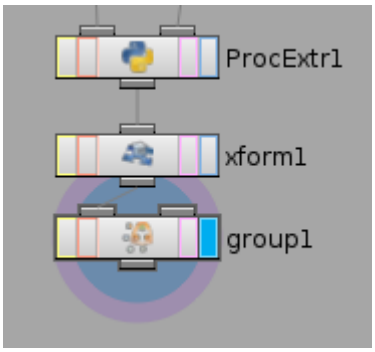


Figura 9.4 nodes encapsulats dins el HDA

9.4. Mètodes implementats

En aquesta secció es detallarà com han estat implementades les funcions més complexes que s'han explicat anteriorment, en el capítol 8 anàlisi i disseny del sistema.

- Com es dibuixa l'edifici un cop es tenen les estructures de dades, dins la classe ProceduralExtrusions. Veure figura 9.4.1.

```
def convertBuilding(self,estructures, geo):
    vertexs=estructures[0]
    cares=estructures[1]
    for c in cares:
        poly = geo.createPolygon()
        for id in c:
            position = vertexs[id]
            point = geo.createPoint()
            point.setPosition(position)
            poly.addVertex(point)
```

Figura 9.4.1 Implementació del mètode convertBuilding

Observant el pseudocodi de la Figura 9.4.1, primer es fa un bucle per recórrer cada cara, un cop a dins es crea un objecte polígon de Houdini i llavors es fa un altre bucle per recórrer tots els vèrtexs que té la cara. D'aquesta manera, cada vèrtex que hi ha dins la cara s'afegeix dins l'objecte polígon, i es va dibuixant l'edifici.

- Com es converteixen tots els perfils a l'estructura de dades adequada pel CampSkeleton, dins la classe Wrapper. Veure Figura 9.4.2.

```
private Profile[] makeProfiles(float perfils [][][]){
    Profile newProfiles[] = new Profile[perfils.length];
    for(int p=0;p<perfils.length;p++){
        float perfil[][] = perfils[p];

        Profile profile = new Profile();
        Loop<Bar> loop;
        profile.points.add(loop = new Loop());

        for(int i=0;i<perfil.length-1;i++){
            Point2d pa= new Point2d(perfil[i][0],perfil[i][1]);
            Point2d pb= new Point2d(perfil[i+1][0],perfil[i+1][1]);
            loop.append(new Bar (pa,pb));
        }
        newProfiles[p] = profile;
    }
    return newProfiles;
}
```

Figura 9.4.2 Implementació del mètode makeProfiles

Observant el pseudocodi de la figura 9.4.2, primer de tot s'inicialitza una llista de Profiles, els perfils ja convertits. Llavors es recorren tots els perfils que hi hagi, i per cada un es converteix a l'estructura adequada pel CampSkeleton, o sigui utilitzant l'objecte Bar que representa un costat.

- Com es fa la reconversió de l'estructura de dades de CampSkeleton. Veure figura 9.4.3.

```
public void emplenarEstructura(){       //reconvertir estructura de dades
    cares = new int[s.output.faces.size()][]; //init
    int nCares=0;
    int nVertexs=0;;
    for (Face f : s.output.faces.values()){     //PER CADA CARA
        LoopL<Point3d> l = f.getLoopL();

        //EMPLENANT ESTRUCTURES DE DADES
        for (Loop<Point3d> pts : l) {
            int vertexXCara=0;
            cares[nCares]=new int[l.count()];
            for (Point3d pt : (pts)) {
                float taula [] = new float[3];
                taula[0]=(float)pt.x;
                taula[1]=(float)pt.y;
                taula[2]=(float)pt.z;

                if(!existeix(taula)){
                    vertexs.put(nVertexs, taula);
                    cares[nCares][vertexXCara]=nVertexs;
                    nVertexs++;
                }
                else{
                    cares[nCares][vertexXCara]=trobarVertex(taula);
                }
                vertexXCara++;
            }
        }
        nCares++;
    }
}
```

Figura 9.4.3 Implementació del mètode emplenarEstructures

Observant el pseudocodi de la figura 9.4.3, s'ha de realitzar un bucle per recórrer cada cara, i un altre per recórrer cada vèrtex de la cara. De manera que cada vegada que es recórrer a una nova cara, es crea una nova posició a la llista de cares, i cada vegada que es recorre un vèrtex de la cara, primer es mira si ja existeix dins la llista de vèrtexs, si ja existeix no es modifica res de la llista de vèrtexs, i en la llista de cares s'introdueix la posició on es troba el vèrtex dins la llista de vèrtex. Si no existeix el vèrtex, es crea una nova posició a la llista de vèrtexs i a la llista de cares es guarda aquesta nova posició on s'ha guardat el vèrtex.

10. Resultats

En aquest capítol es mostraran els resultats obtinguts seguint la metodologia descrita en el capítol 3. Aquests resultats estan il·lustrats a partir de les imatges de la zona de treball de Houdini.

10.1. Edifici inicial

En aquest primer exemple es mostrarà el resultat en generar un edifici partint d'una planta i un únic perfil relativament senzills geomètricament.

S'han dissenyat la planta i perfil que es mostren a la Figura 10.1.

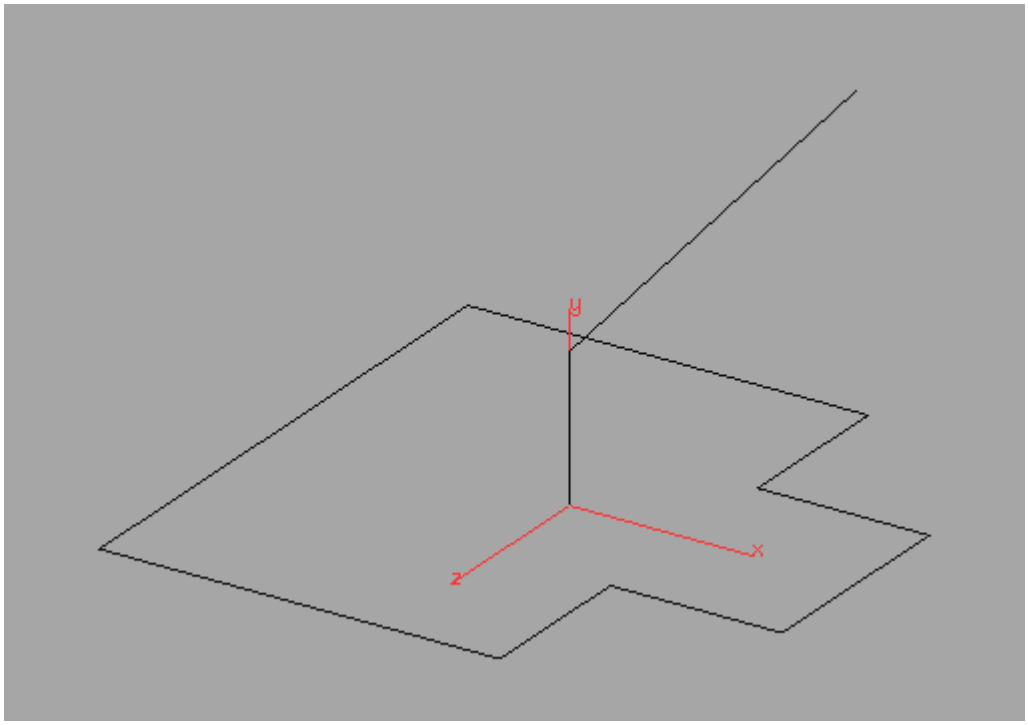


Figura 10.1 planta i perfil del exemple "edifici senzill"

Amb els valors:

Planta: [2,0,-2] [5,0,-2] [5,0,2] [2,0,2] [2,0,5] [-5,0,5] [-5,0,-5] [2,0,-5]

Perfil: [0,0,0] [0,2.5,0] [5,8,0]

Per tant es veu que al aixecar l'edifici, primer s'alçarà fins a 2.5 totalment vertical, i llavors

cada vèrtex s'aproparà al centre seguint la direcció indicada definida al perfil. A la Figura 10.2 es mostra el resultat.

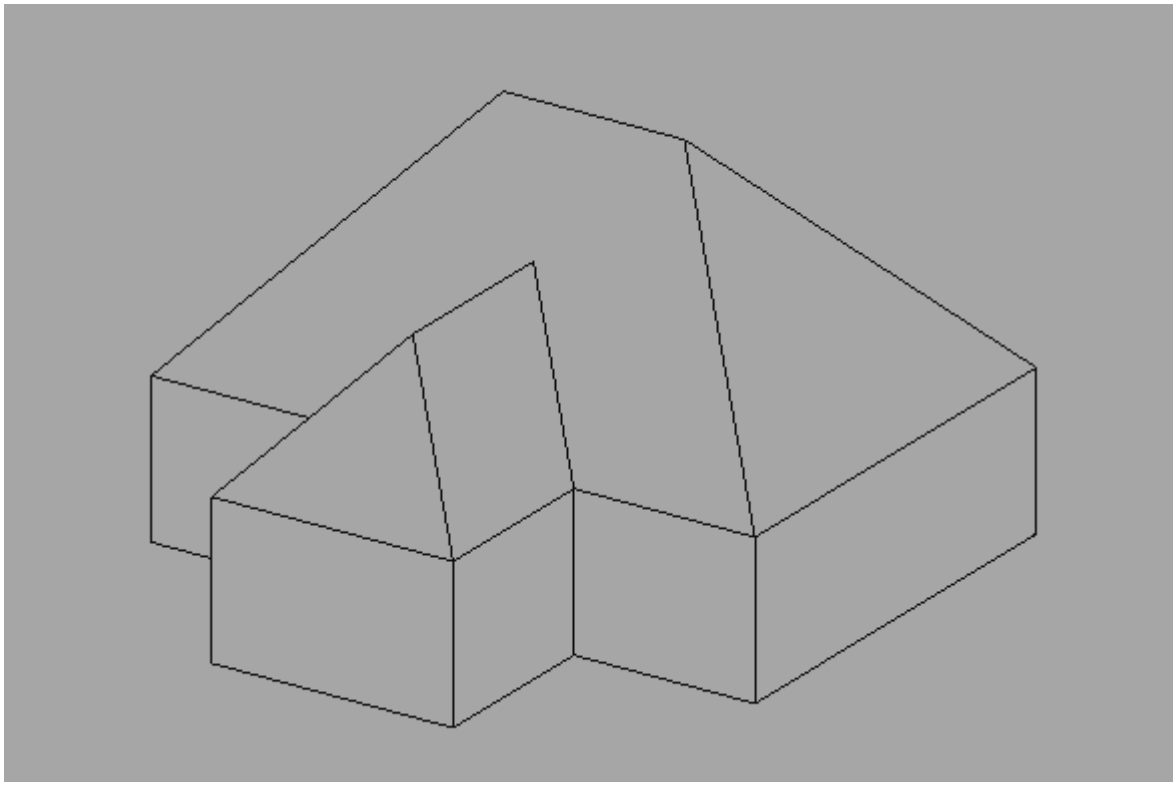


Figura 10.2 Resultat de l'edifici senzill

10.2. Edifici amb múltiples perfils

S'han dissenyat la planta i els perfils que es mostren a la figura 10.3. Amb els següents valors:

Planta: [6,0,0] [6,0,6] [0,0,6] [0,0,0] [-6,0,0] [-6,0,-6] [6,0,-6]

Perfil1: [0,0,3] [0,6,3] [2,8,3]

Perfil2: [0,0,-6] [0,6,-6] [-1,6.2,-6] [3,12,-6

perfil3: [3,0,6] [3,6,6] [1,6.2,6] [5,8,6]

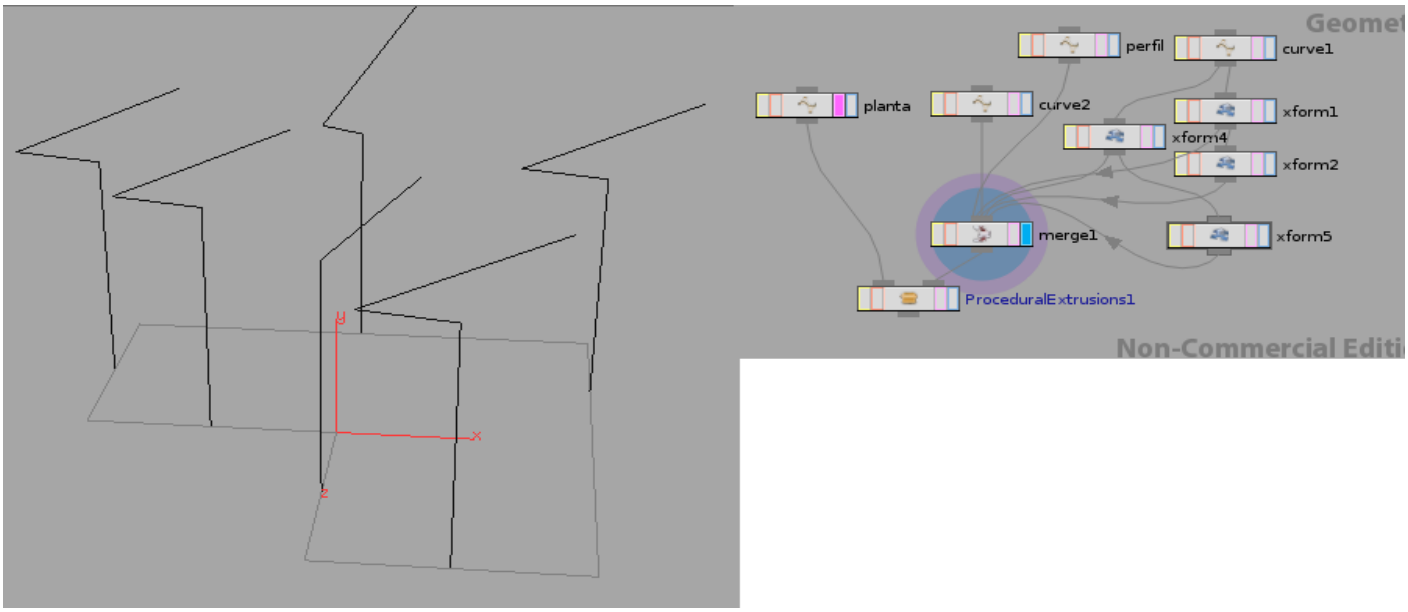


Figura 10.3 Definició de la planta i perfils

Observant la Figura 10.3, a l'esquerra es mostren els perfils juntament amb la planta, de manera que es veu cada perfil a quines façanes influirà. A la dreta es veu l'arbre de nodes, i s'hi pot observar com realment s'han definit 3 perfils, encara que en surtin 6 a la figura, degut a que s'han realitzat translacions d'un mateix perfil perquè s'associés amb més d'una façana. El resultat de l'edifici es pot veure a les Figures 10.4 i 10.5.

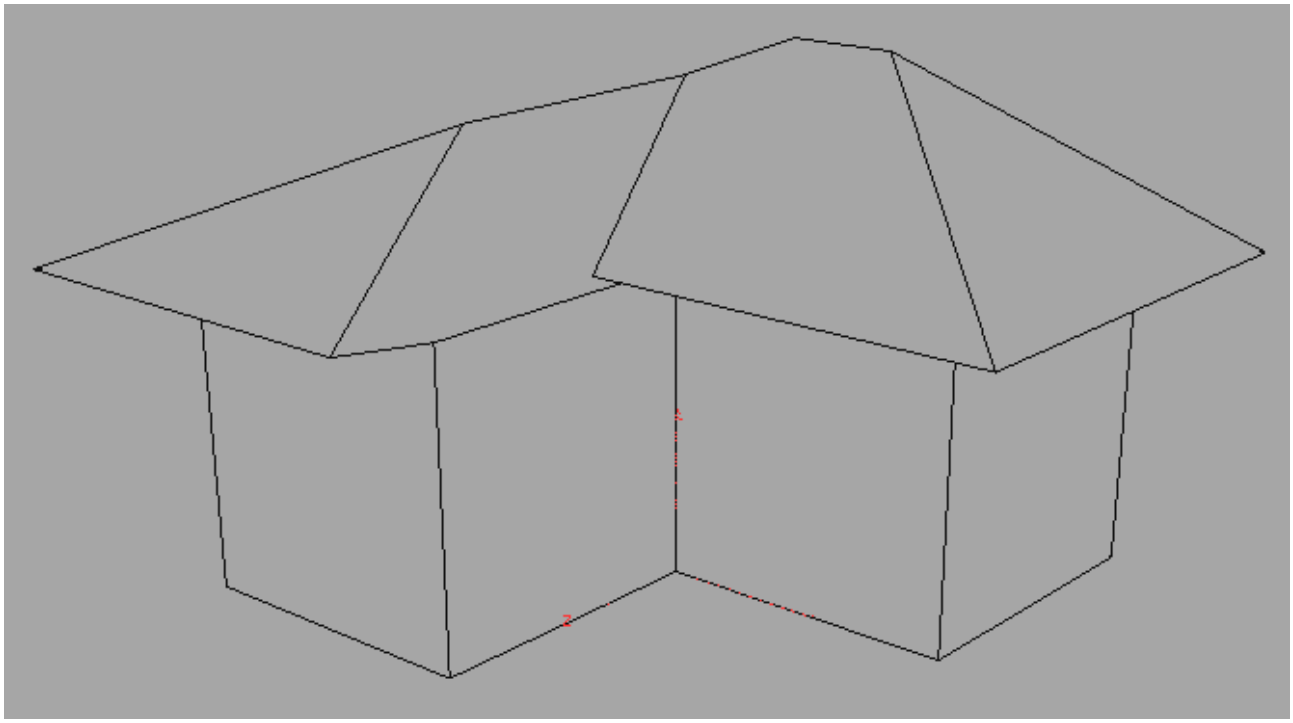


Figura 10.4 Resultat de l'edifici (I)

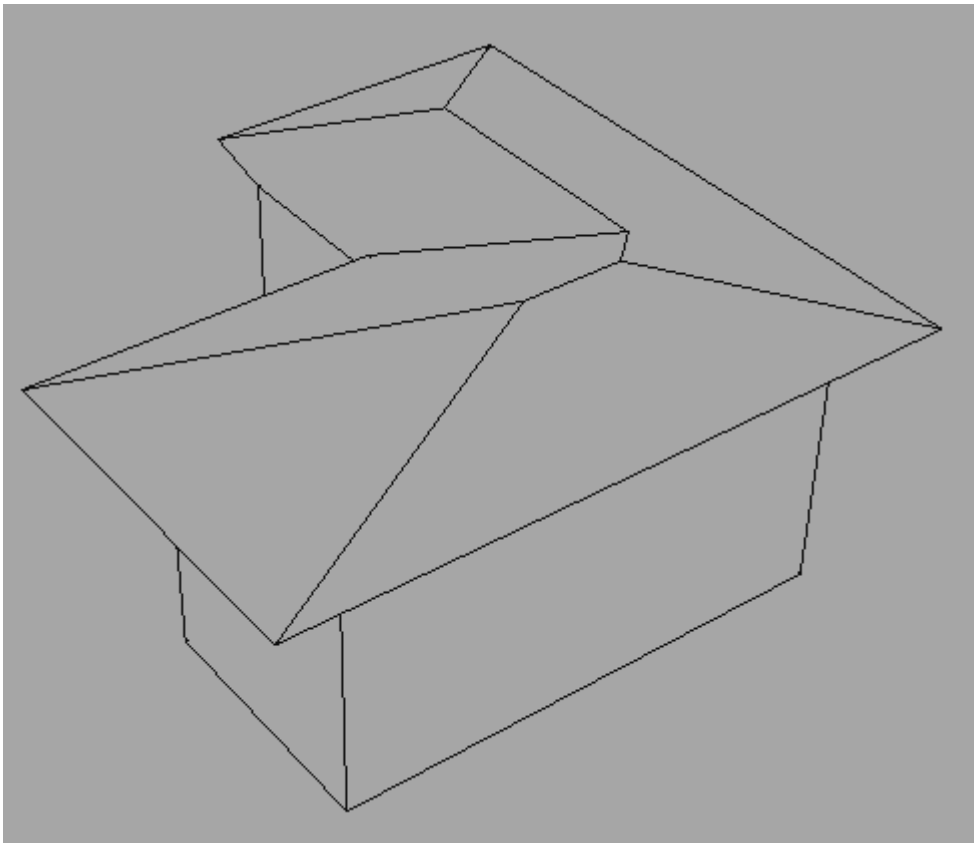


Figura 10.5 Resultat de l'edifici (II)

10.3. Edifici integrat amb buildingEngine

S'han dissenyat la planta i els perfils que es mostren a la figura 10.6. Amb els següents valors:

Planta: [7,0,3] [5,0,4] [5,0,10] [-6,0,5] [-6,0,2] [-6,0,-0.5] [-6,0,-2] [-3,0,-2] [-3,0,-9] [5,0,-9] [5,0,-4] [7,0,-3]

Perfil1: [-4,0,-3] [-4,7,-3] [-5.5,7.2,-3] [-1,11,-3]

Perfil2: [7,0,0] [7,7,0] [6.5,7.2,0] [10,11,0]

perfil3: [0,0,7.5] [0,7,7.5] [-2,7.2,7.5] [1,11,7.5]

Perfil4: [0,0,-9] [0,7,-9] [-1,7.2,-9] [1,8,-9]

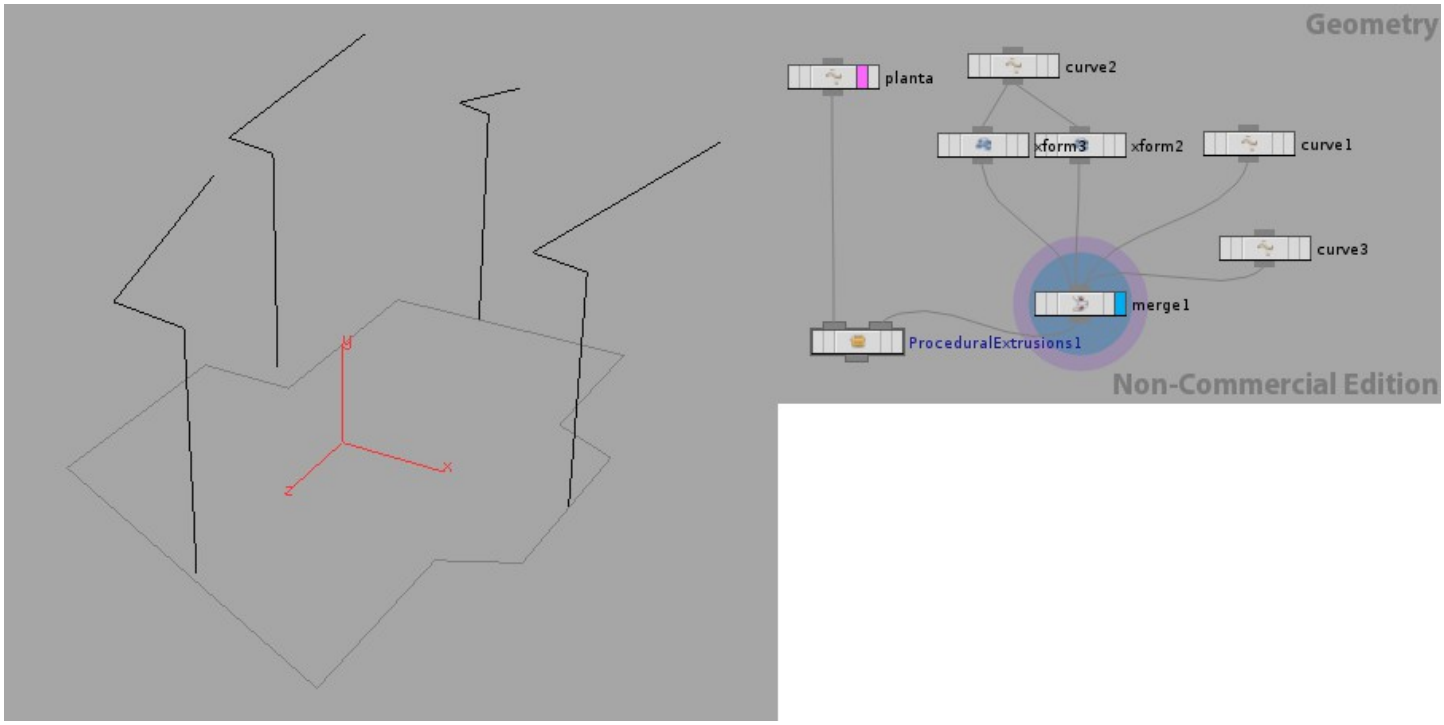


Figura 10.6 Definició de la planta i perfils

Observant la Figura 10.6, a l'esquerra es mostren els 4 perfils juntament amb la planta, de manera que es veu cada perfil a quines façanes influirà. A la dreta es veu l'arbre de nodes, i s'hi pot observar com realment s'han definit 3 perfils, però com que es volia utilitzar un mateix perfil en 2 façanes allunyades entre elles, s'han fet 2 translacions a aquesta "curve" perquè quedin aquests 2 perfils molt a prop a les 2 façanes respectivament. El resultat de l'edifici es pot veure a les Figures 10.7, 10.8 i 10.9.

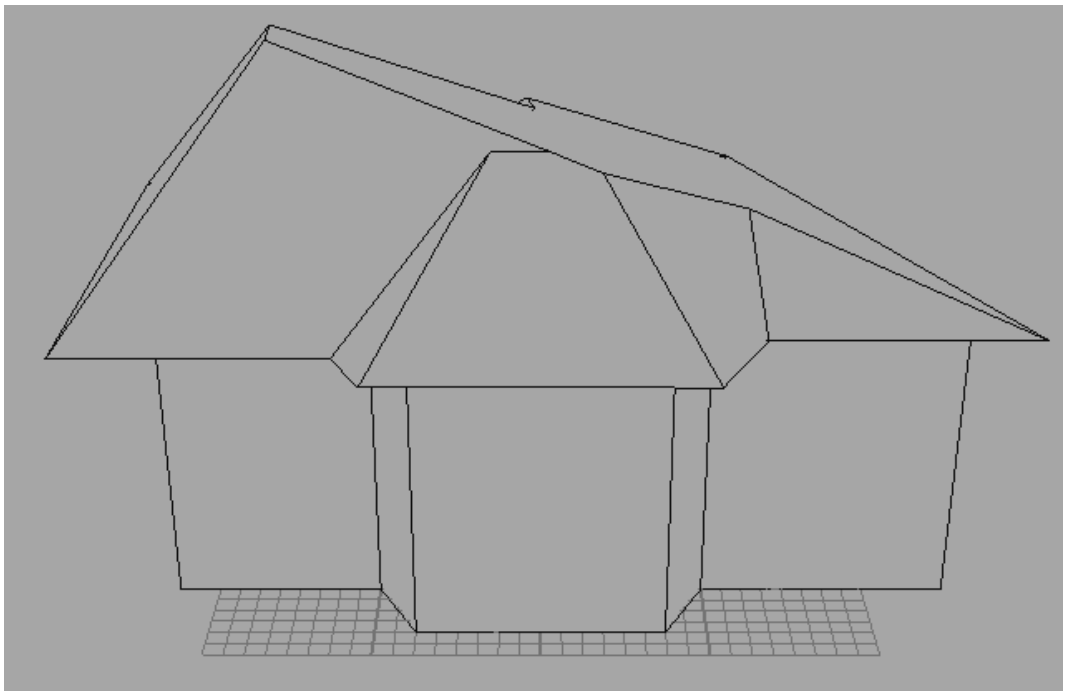


Figura 10.7 Resultat de l'edifici (I)

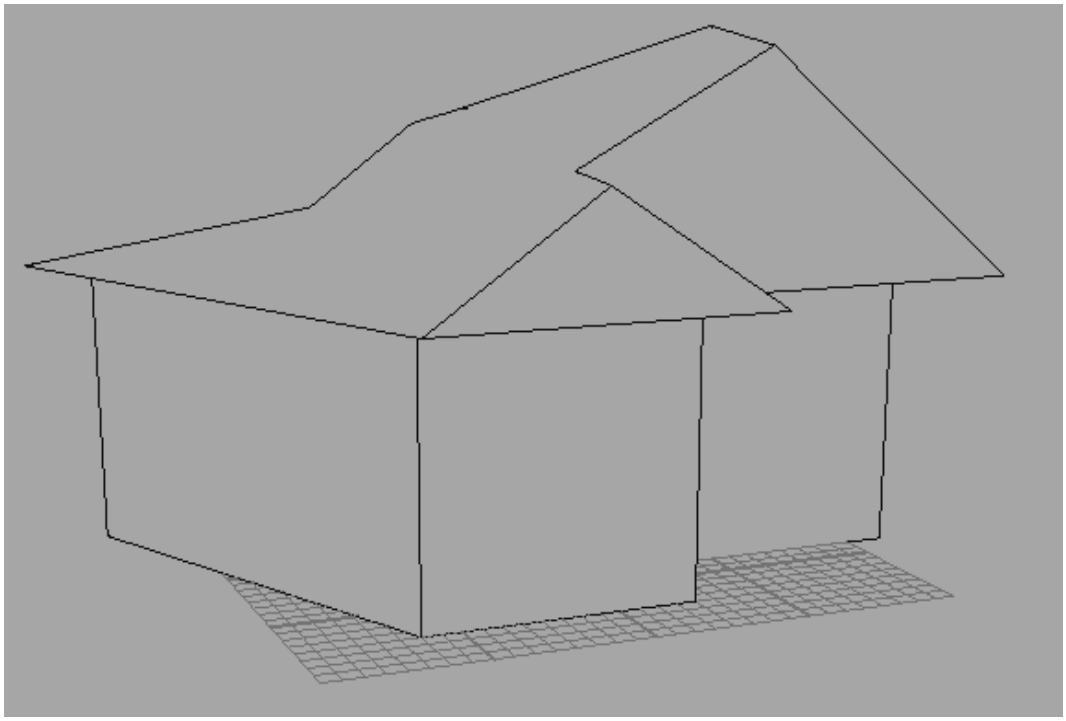


Figura 10.8 Resultat de l'edifici (II)

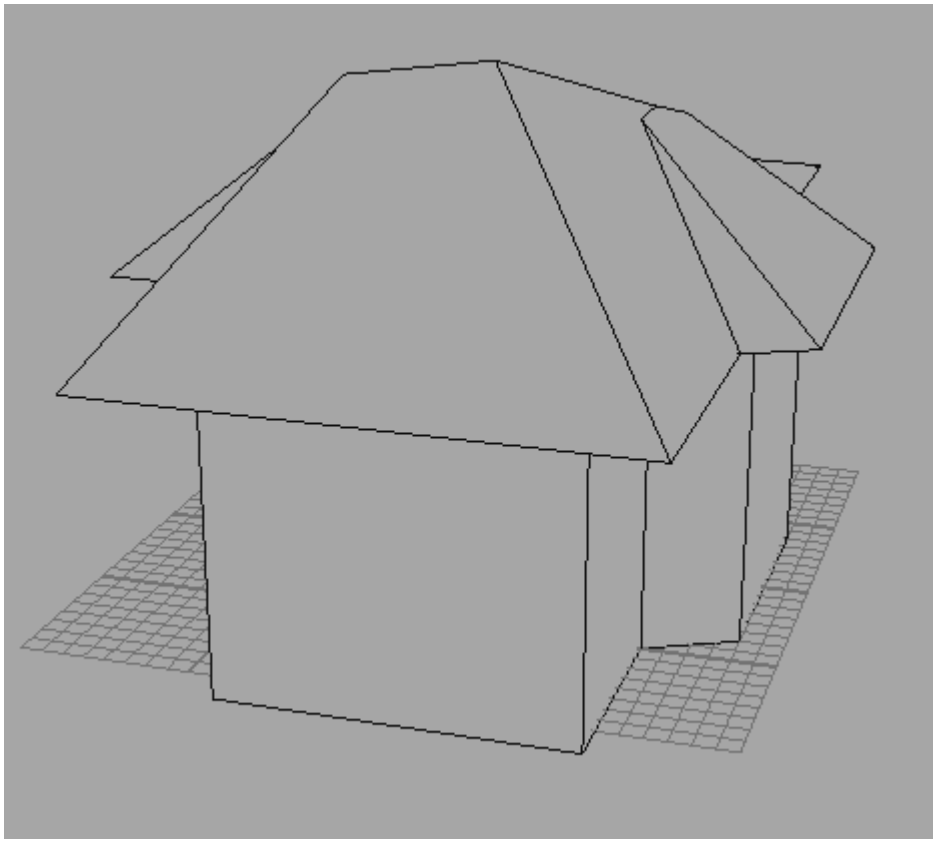


Figura 10.9 Resultat de l'edifici (III)

A continuació s'ha realitzat la integració amb buildingEngine, de manera que s'han afegit finestres i portes. Veure figures 10.10, 10.11, 10.12 i 10.13.

Figura 10.10 Integració amb buildingEngine (I)

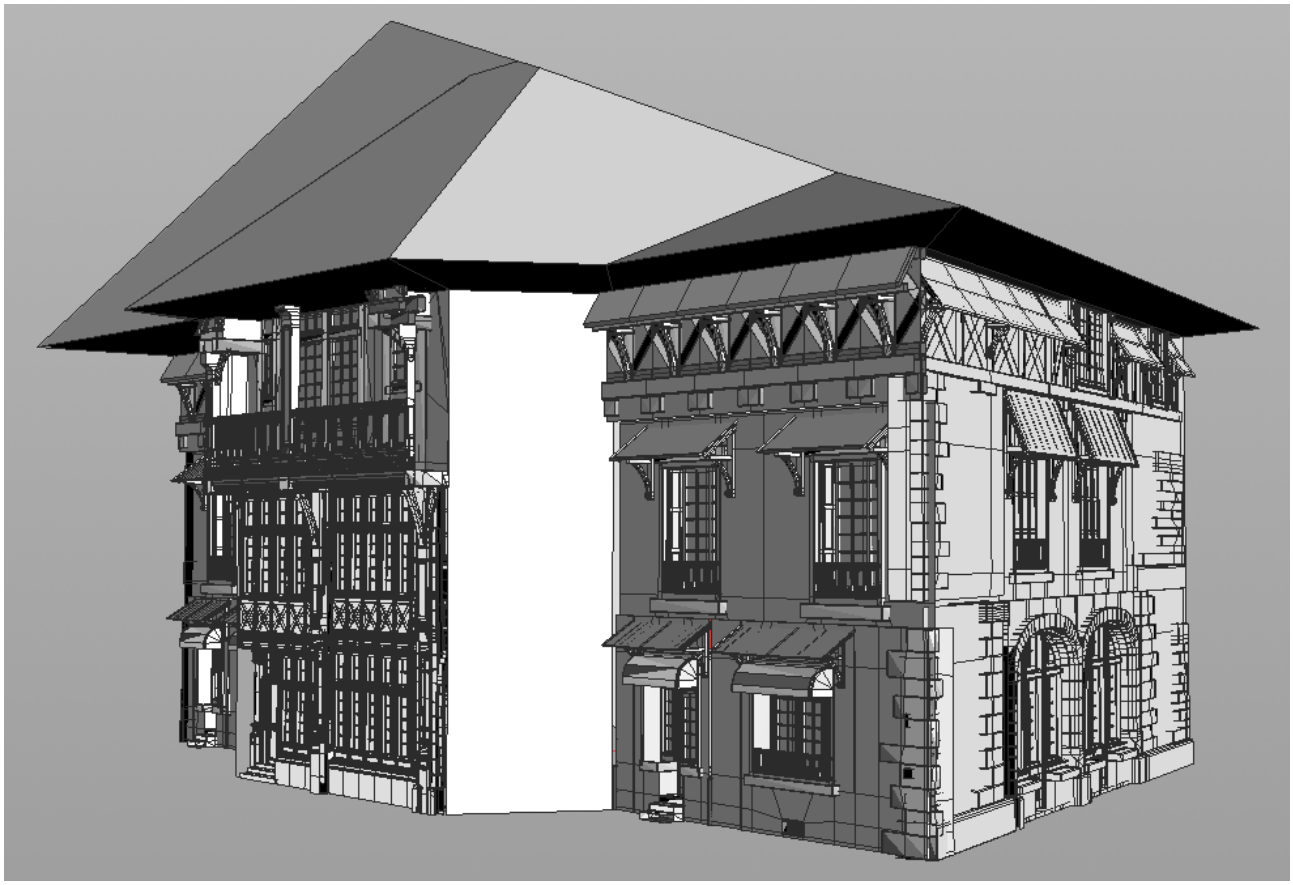


Figura 10.11 Integració amb buildingEngine (II)

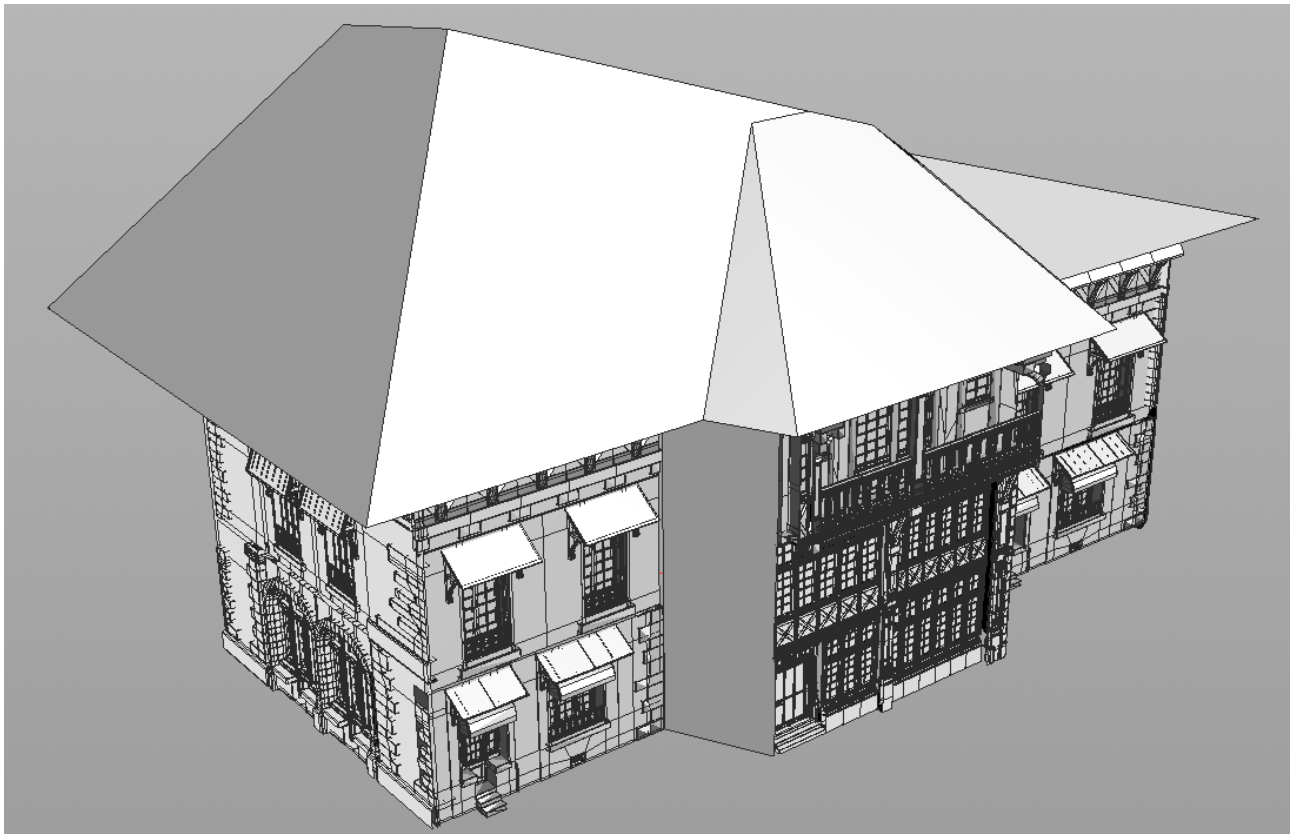


Figura 10.12 Integració amb buildingEngine (III)

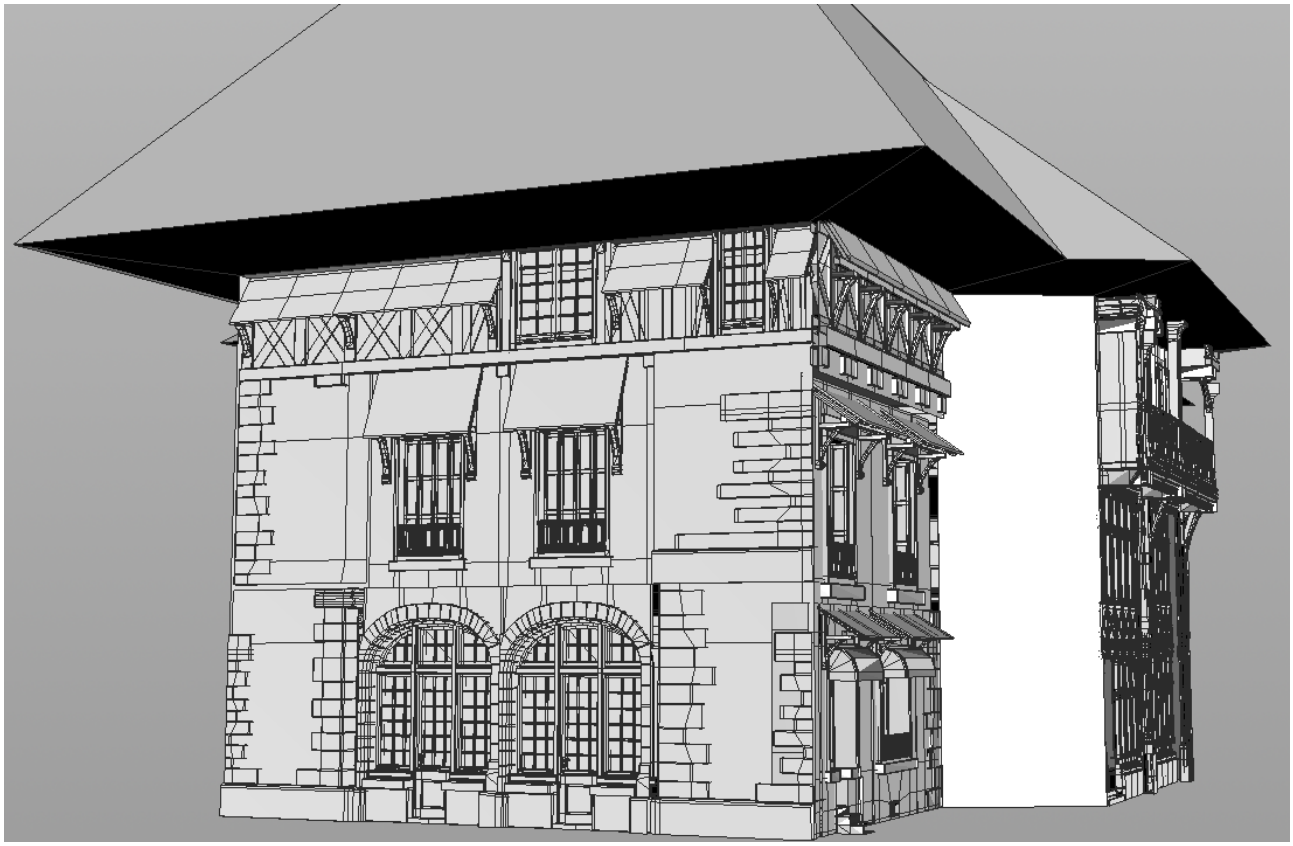


Figura 10.13 Integració amb buildingEngine (IV)

10.3.1. Integració amb buildingEngine (segona versió)

En aquesta secció es treballa amb el mateix edifici que s'ha creat en el capítol anterior, 10.3, però s'han introduït uns altres accessoris de buildingEngine. Veure figures 10.14, 10.15, 10.16 i 10.17.

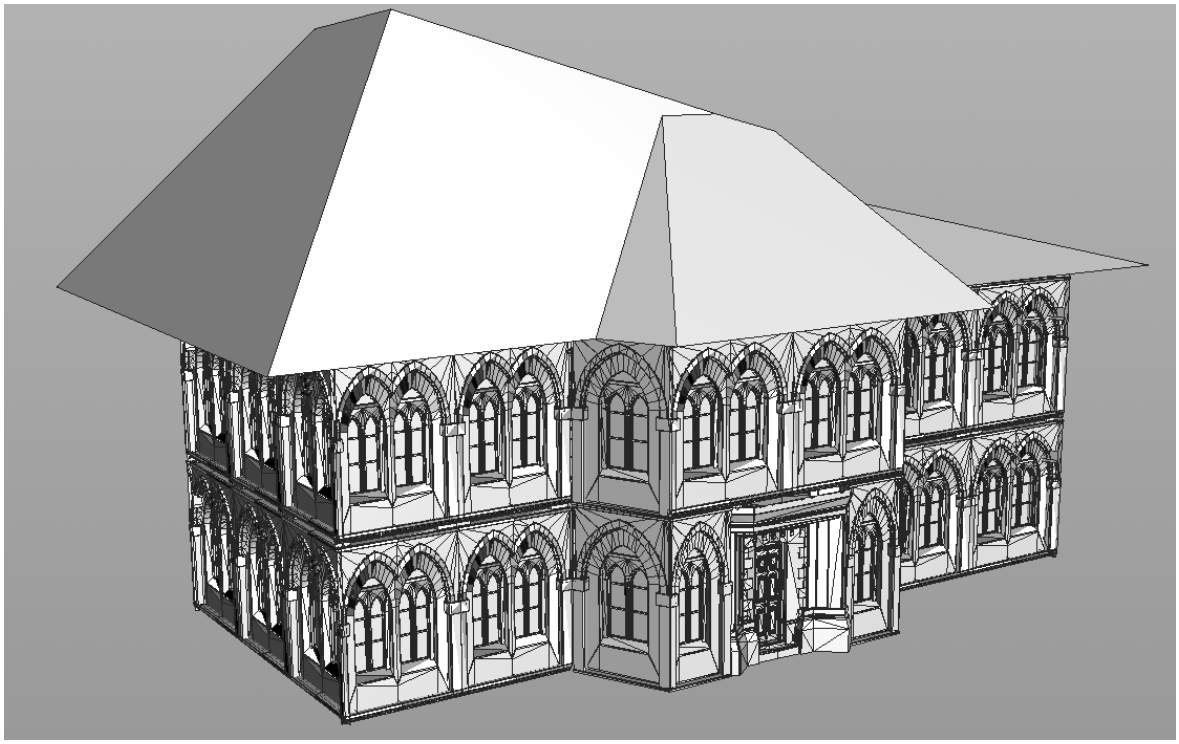


Figura 10.14 Integració amb buildingEngine (segona versió) (I)

Figura 10.15 Integració amb buildingEngine (segona versió) (II)

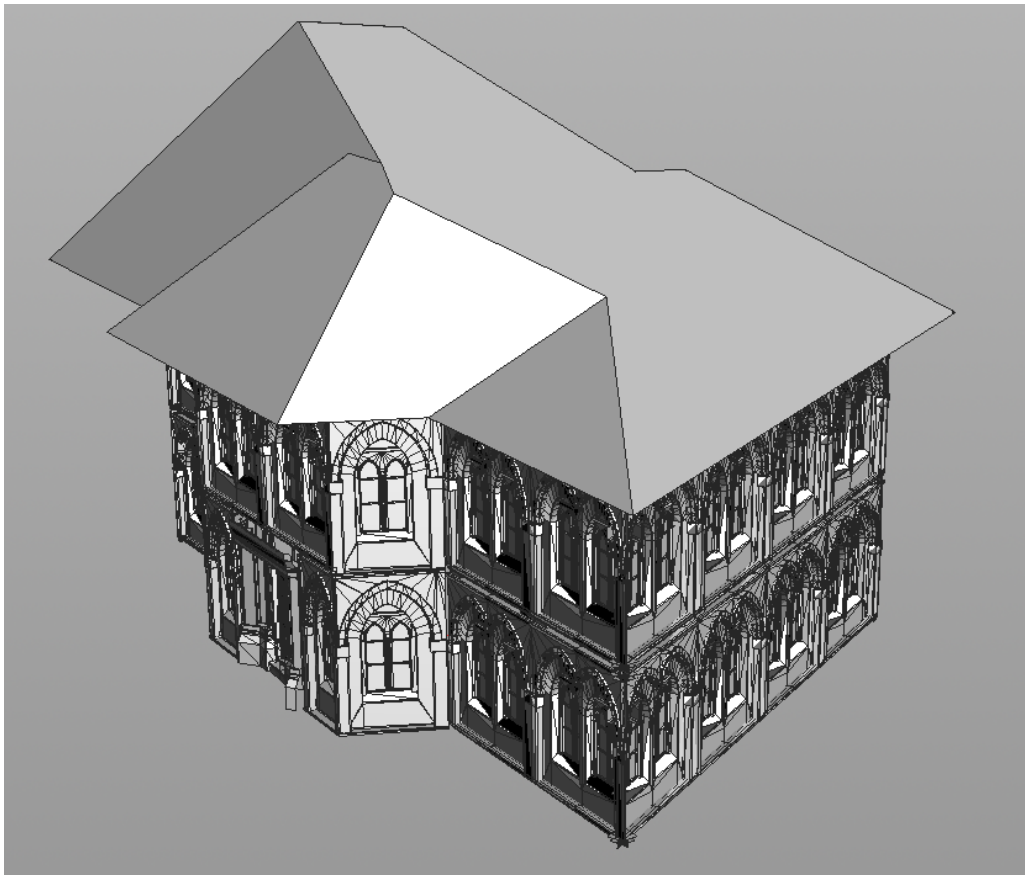


Figura 10.16 Integració amb buildingEngine (segona versió) (III)

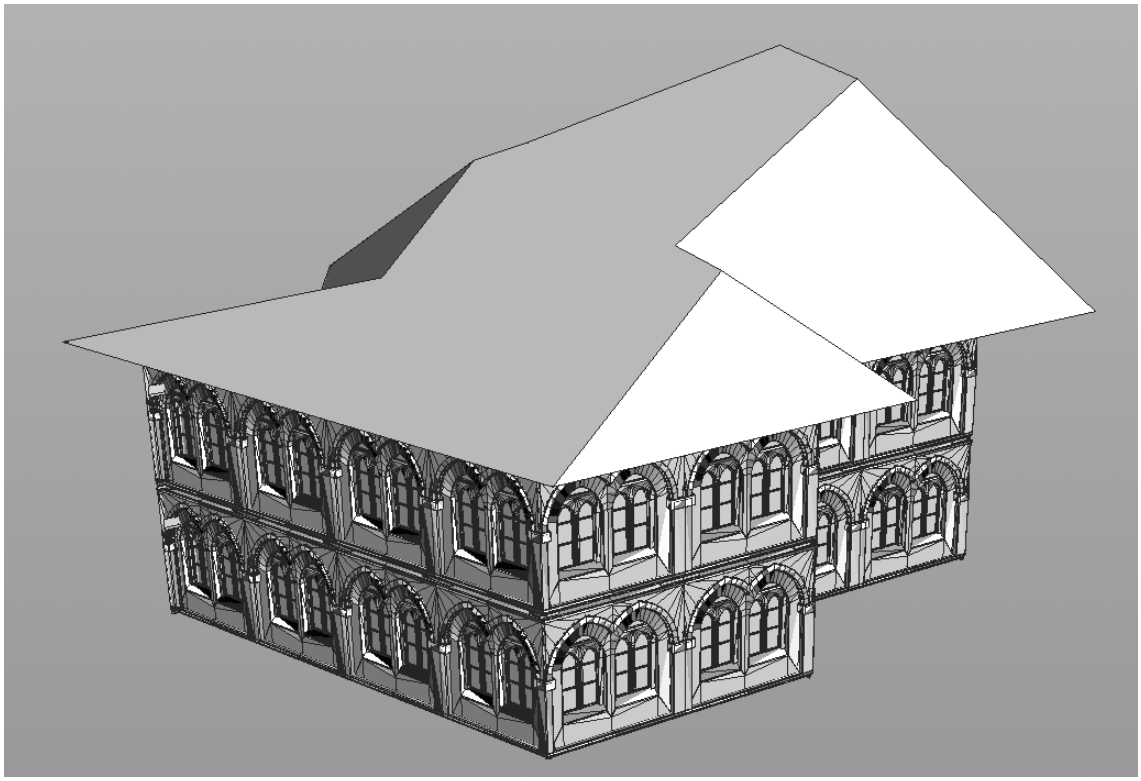


Figura 10.17 Integració amb buildingEngine (segona versió) (IV)

11. Conclusions

Una vegada finalitzada tota la feina plantejada a l'inici de l'elaboració d'aquest projecte, és moment de valorar el treball realitzat durant aquests mesos.

L'objectiu bàsic d'aquest projecte és el desenvolupament d'una eina de fàcil ús, que permeti a l'usuari obtenir de forma àgil i ràpida un edifici procedural.

S'ha creat una eina que, un cop executat l'algorisme, genera la geometria bàsica d'un edifici. El fet que es pugui seguir manipulant amb el **skyline**Engine fa que sigui una bona eina a l'hora de crear els edificis amb tots els accessoris, portes, finestres, etc.

Durant la elaboració d'aquest projecte s'han arribat a varies conclusions de les quals destaco:

- És fonamental utilitzar les eines adequades i realitzar un bon anàlisi dels problemes que poden sorgir. Encara que s'estigui més temps en arribar a una millor solució, es nota en el resultat final i s'arriben a solucionar millor els problemes.
- S'ha après a programar en Python. És un llenguatge fàcil d'aprendre, molt potent, ràpid d'implementació i que fa que el codi quedi bastant net.
- S'ha après a utilitzar l'entorn de Houdini i a programar Python SOP.
- S'ha après a utilitzar eines útils que no estaven dins el pla del projecte inicial, com ara el xmlrpclib i SimpleXMLRPCServer.

11.1. Temporització

Poques vegades la temporització planificada inicialment en un projecte correspon fidelment amb el que s'obté en la realitat. La temporització real del temps dedicat al projecte es tal com es mostra a la Figura 11.1.

Originalment s'havia plantejat acabar el projecte al Juliol, però a causa d'uns problemes tècnics causats per la llibreria Jpype, no es va poder arribar al objectiu inicial. Per tant el temps d'implementació, tant de Java com de Python, i de integració es van allargar notablement.

Figura 11.1 Comparació: a l'esquerra planificació inicial, a la dreta el desenvolupament real

12. Treball futur

El treball realitzat en aquest projecte permet un conjunt de millores i ampliacions. Tant es podrien millorar les funcionalitats del projecte com implementar-ne de noves. Tot seguit es presenten unes possibles idees que es podrien dur a terme.

Una millora que s'hi podria aplicar és adaptar l'edifici al terreny. És a dir, que s'emmotllés a un terreny muntanyós per exemple, o simplement amb un terreny amb una certa inclinació, etc.

Una altra millora a introduir podria ser el crear decoració per aquestes construccions aplicant textures a la façana dels edificis.

13. Bibliografia

- Houdini 11 Documentation. 2011. Side Effects Software. 2012
  http://www.sidefx.com/docs/houdini11.1/

- Tom Kelly and Peter Wonka. (2011). Interactive architectural modeling with procedural extrusions. *ACM Trans. Graph.* 30, 2, Article 14 (April 2011), 15 pages.
  http://dl.acm.org/citation.cfm?doid=1944846.1944854

- Python para todos. Raúl González Duque. 20 Novembre 2011.
  https://launchpadlibrarian.net/18980633/Python%20para%20todos.pdf

- Wikipedia, 2011. Jimmy Wales. 2012
  http://ca.viquipedia.org
  http://es.wikipedia.org
  http://en.wikipedia.org

- Web del projecte **skyline**Engine
  http://ggg.udg.edu/skylineEngine/

14. Annexos

- Interactive Architectural Modeling with Procedural Extrusions

Interactive Architectural Modeling with Procedural Extrusions

TOM KELLY
University of Glasgow
and
PETER WONKA
Arizona State University

We present an interactive procedural modeling system for the exterior of architectural models. Our modeling system is based on procedural extrusions of building footprints. The main novelty of our work is that we can model difficult architectural surfaces in a procedural framework, e.g. curved roofs, overhanging roofs, dormer windows, interior dormer windows, roof constructions with vertical walls, buttresses, chimneys, bay windows, columns, pilasters, and alcoves. We present a user interface to interactively specify procedural extrusions, a sweep plane algorithm to compute a two-manifold architectural surface, and applications to architectural modeling.

Categories and Subject Descriptors: I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*Modeling packages*

Additional Key Words and Phrases: procedural modeling, roof modeling, urban modeling

ACM Reference Format:

Kelly, T. and Wonka, P., Interactive Architectural Modeling with Procedural Extrusions. ACM Trans. Graph. VV, N, Article XXX, XX pages.
DOI = 10.1145/XXXXXXX.YYYYYYY

1. INTRODUCTION

The main motivation for our work is to develop an *interactive* and *procedural* modeling tool for *complex* architectural surfaces. We are interested in procedural and interactive modeling for three reasons. First, procedural descriptions allow edits to architectural surfaces at multiple levels and previous edits will adapt to subsequent ones. For example, the scene in Fig. 2 can be edited by reshaping the building footprints, and the model buildings, including the complete roof construction, will change according to the new input. Second, procedural modeling is the most efficient method to generate larger urban environments. Finally, we want to combine interactive and procedural modeling, because a frequent obstacle to using procedural tools is that it requires scripting. Eliminating scripting will enable more people to use procedural modeling tools.

Fig. 1. Procedural extrusions allow a footprint (2d plan) to be extruded to form the walls and roof of a house (inset). Meshes and procedural details can then be attached (main).

Our goal is to model complex architectural features, including overhanging roofs, dormer windows, interior dormer windows, roof constructions with vertical walls, buttresses, chimneys, bay windows, columns, pilasters, and alcoves. See Fig. 1 for an example showing some of these features. These complex architectural surfaces have not been handled in procedural modeling before, and the main contribution of this paper is to introduce the first procedural modeling solution that includes these surfaces. Previous work in procedural modeling using shape grammars [Müller et al. 2006; Lipp et al. 2008] is able to model some architectural roof surfaces on a restricted set of footprints, but not the more complex roofs of arbitrary footprints shown in this paper.

The first part of our solution is to identify the most important edits and to design a user interface to specify procedural extrusions.

Authors' addresses: T. Kelly, Department Of Computing Science, Lilybank Gardens, Glasgow, UK; email: twakelly@gmail.com; P. Wonka, Arizona State University, Tempe, USA; email: pwonka@gmail.com. We would like to thank the reviewers and Michael Schwarz for many helpful suggestions for improving the various drafts of the paper. The research was supported by NSF, the FIT-IT program from FFG, and the Scottish Informatics and Computer Science Alliance.

DOI 10.1145/XXXXXXX.YYYYYYY
http://doi.acm.org/10.1145/XXXXXXX.YYYYYYY

Fig. 2. We present an interactive procedural modeling system that is able to model difficult architectural surfaces, such as roof constructions. This figure shows procedural extrusions applied to 6000 floorplans from a GIS database of Atlanta.

We consider this part interesting because after analyzing examples, such as the one shown in Fig. 1, it is not clear how to model such a building, and what editing operations are even necessary to ensure that a larger class of interesting architecture can be modeled. An important aspect of our solution is to model buildings from floorplans and profile curves, see Fig. 5. In Sec. 3 we will describe our user interface in more detail including the architectural configurations that motivated the different user interface parts. The goal of our work is to have tools that are expressive enough to be able to quickly model most aspects of a building. We will evaluate our system on a catalog of 50 buildings in various styles in Sec. 6 to demonstrate the efficiency of our tools and to document geometric configurations that are difficult to reproduce.

The second part of our solution is a collection of algorithms to compute procedural extrusions from the user specification, see Sec. 4. We propose a sweep plane algorithm to grow the architectural surface upwards and to handle various events stemming from user edits or plane intersections. Our algorithms are inspired by the straight skeleton [Aichholzer et al. 1995]. We want to note that the computational geometry community emphasizes provably correct algorithms and therefore often favors rational arithmetic. In contrast, our work consists of heuristic algorithms that emphasize computation speed and are geared towards a floating point implementation. While our heuristics include various mechanisms to make the results more robust, it is possible that the computations can fail. For example, in the Atlanta data set of 6000 buildings we noted that two roof planes were not computed correctly. The approximate nature of our floating point computation also results in roof planes being moved by millimeters.

The contributions of our work are:

—the design of the system and the set of tool choices to enable procedural modeling of complex architectural surfaces.

—heuristic algorithms to generate a polygonal mesh from the user specification that is approximately consistent with the input data.

—the evaluation of the system on a collection of examples to verify its practical utility, and to identify configurations that are difficult to model with our tools.

2. RELATED WORK

In architecture, Stiny pioneered the idea of shape grammars [Stiny 1975]. In computer graphics grammars were used as a design tool

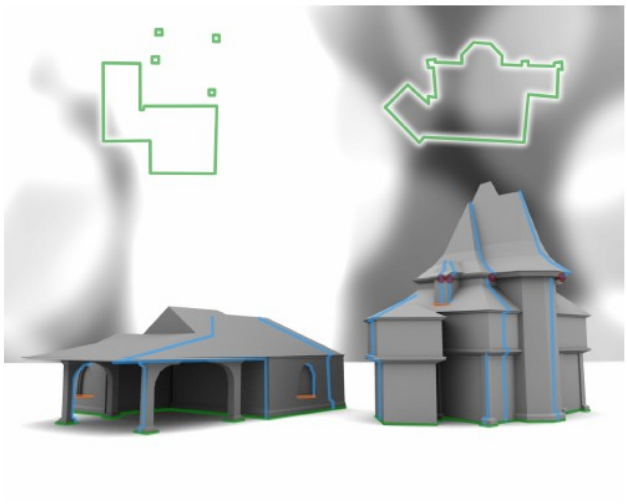


Fig. 3. These two examples show architectural surfaces overlayed with the user input. Plans (green), profiles (blue), natural steps (orange) and offset events (red) are specified in the user interface. The output of our system is an architectural shell (gray).

for architecture by Wonka et al. [2003], Müller et al. [2006], Aliaga et al. [2007], and Lipp et al. [2008]. L-systems [Prusinkiewicz and Lindenmayer 1991] were also proposed for procedural modeling of architecture [Marvie et al. 2005]. Merrel and Manocha [2008] propose a more general approach that can create new models from an example mesh. Given a man-made model as input, great results for reshaping were achieved by Cabral et al. [2009] and Gal et al. [2009]. The output of our procedural extrusions could also be further processed to distribute brick patterns [Legakis et al. 2001].

The straight skeleton was introduced by Aichholzer et al. [1995; 1996] and the authors commented how the straight skeleton computes a very plausible roof construction over a polygon. The idea of weights for the straight skeleton has been briefly mentioned by Eppstein and Erickson [1998], but the topic was only developed for a convex polygon decomposition [Aurenhammer 2008]. These papers are the inspiration for our work, and they aim to make a contribution to theory in computational geometry. In contrast, we focus on the application to modeling and extensions that are inspired by the demands of our modeling system. A starting point for

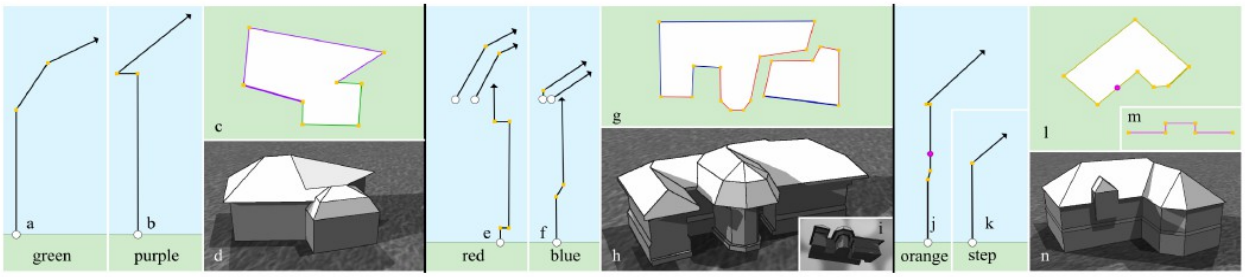


Fig. 4. Three example buildings constructed in our user interface. We demonstrate multiple profiles on a simple plan (abcd), modeling overhangs (efghi) and anchors (jklmn). Simple profiles (ab) are applied to the green and purple edges of the plan (c) to create the geometry (d). Note the horizontal profile section. Overhangs are defined using an additional pair of profile polylines associated with every edge (ef) to create typical roof geometry (hi). Anchors (magenta circles) are defined on the profile (j) and the plan (l) to position features. In this example the anchors position a rectangular natural step (m) with a profile (k) that creates a roof-window (n).

our implementation was the work by Felkel and Obdrzalek [1998] and Cacciola [2004].

Applications to architectural modeling of extrusion operations and the straight skeleton were demonstrated by several authors, e.g. [Laycock and Day 2003; Havemann 2005; Müller et al. 2006; Kelly 2006; Autodesk Inc.]. Our goals are similar to these approaches and we contribute new extensions to the straight skeleton and an interactive procedural modeling system.

3. USER INTERFACE DESCRIPTION

Our interface controls a sequence of extrusions that are particularly suitable for creating the shell of architectural models. In this section we will introduce the functionality of our user interface.

3.1 Modeling With Profiles

The inspiration for our work comes from simple roofs, that are defined by a 2d polygonal floorplan and an angle that defines the roof slope. We extend this simple concept to construct a wide variety of roofs by using different angles on each edge and entire building shells by changing the angle as we ascend using *profiles*, Fig. 3.

To construct our geometry we use a sweep plane that rises vertically from the base of the building. This sweep plane defines an *active plan* that combines the changing profiles, and discrete modifications to create complex architecture.

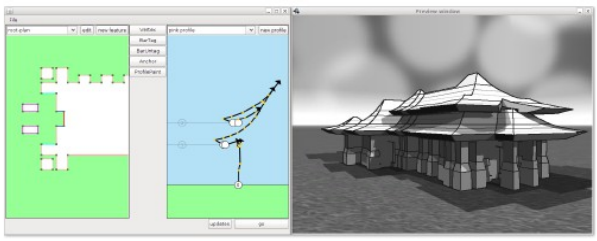


Fig. 5. The interactive interface during the design of a temple. The right window contains the output preview whilst the left window contains the plan and the profile editors.

3.2 Plans and Profiles

The UI consists of one pane showing the plan, one pane for profile, and a 3d preview window as shown in Fig. 5.

The plan is a set of edges (see Sec. 4.1 for the definition of a plan). For each edge in the plan, there is an associated collection of polyline segments, called a *profile*, that define the shape of a cross-section through the building at that plan edge. As the user edits either the plan or the profiles, our system shows the resulting architectural shell in the 3d preview window.

The user interface presents standard operations for inserting, deleting, and moving vertices in the profile polylines, and vertices (called corners) in the floor plan. In Fig. 4(abcd) we show an example with one polygon as the plan (c) and two profiles (a and b). The plan edges are color coded to show which of the two profiles is associated with each edge. Note that the profiles have to be monotonic in the vertical direction, but we allow horizontal polyline segments as special case. In the implementation, every change of direction in a profile will lead to an edge direction event (Sec. 4.5).

3.3 Overhangs

One important design choice we had to make is how to model overhangs. There are two possibilities: 1) Allow the user to draw arbitrary polylines as profiles that can go up or down in the vertical direction; 2) Force the user to explicitly model profiles as multiple polylines where each polyline must be monotonic in the vertical direction. After some experiments we decided that the second option makes it easier to synchronize overhangs across multiple profiles. We will explain the process of modeling overhangs using the second example in Fig. 4 (efhgi).

The user creates the input floor plan shown in (g). The edges in this floor plan are color coded as either red or blue. A red edge will be extruded according to the red profile (e) and the blue edges will be extruded according to the blue profile (f). The final geometric construction is shown in (h) and (i). The red profile as well as the blue profile each consist of three polylines. Each of these polylines is monotone in the vertical direction. Modeling overhangs is an explicit operation. The overhang is modeled by inserting two new polylines into both profiles at a certain height. In the user interface this is one atomic insertion operation. If the user clicks to add a new

vertex overhang in one profile, then all profiles will obtain two new polylines at the same height. The user can edit the new polylines for each profile independently, only the starting height will remain synchronized. In the implementation, we will trigger a profile offset event (Sec. 4.6) at this height.

3.4 Anchors

Several editing operations require us to locate features on the manifold. These might include meshes, such as doors and windows, or discrete changes to the plan, such as chimneys. The features have to be placed so that they can still be located after subsequent edits. This is called the persistence problem in editing procedural models [Lipp et al. 2008], and we introduce *anchors* as a solution in our system.

The user can place anchors by selecting a location in the 3d view, or by selecting points on the input plan's edge and the corresponding profile polyline. In Fig. 4 (jklmn) the anchors are shown as magenta circles on a floor plan edge and a profile edge. We allow the user to select from two types of anchor on a plan edge — relative and absolute. A relative anchor's location is a fraction of its length on the active plan edge. If the edge is represented in the active plan at the specified height, the feature is instanced.

Absolute anchors are defined on an input plan edge, and define a plane perpendicular to this edge. The intersection of this plane and the corresponding edge in the active plan at the height specified by the profile anchor defines the instance location. Because an edge in the active plan may shrink as it ascends, absolute anchors may not be instanced if they lie outside the edge on the active plan. Because an active plan edge may grow, it is possible to position absolute anchors beyond the ends of the input plan edge.

3.5 Plan Edits

Discrete edits to the plan at a certain height are know as *plan edits*. These are located by anchors specified by the user, and may modify, create or delete edges in the active plan. In the example Fig. 4 (jklmn) a plan edit is introduced at the location of the anchor. The plan edit itself is a set of edges (m). These edges are extruded along the new profile (k). Again the user is offered several techniques with different advantages. *Forced steps* insert an arbitrarily set of edges into the plan, while *natural steps* offer a range of simple shapes that can be inserted. For reasons that are discussed later, forced steps are more powerful, but can lead to self intersections, while natural steps are guaranteed to create manifold geometry.

As shown in Fig. 6 we can use discrete plan edits to locate features such as roof windows, or chimneys. Additionally, by adding a rectangle exterior to the active plan, and applying the appropriate profiles, we can create buttresses, as in Fig. 23. If the input plan has several repeated elements, such as bay windows or buttresses, plan edits give a convenient tool for defining an *instance* once, whilst repeating it at a number of different anchored locations.

3.6 Positioning Procedural Details

Anchors can be used to mark the top-left and bottom-right elements in a grid of features, such as windows. Parameters can be set to control the width of the repetitions, and when combined with relative anchors, allow features to be distributed on resizable façades. Variations on this theme allow rows or columns of features to be located,

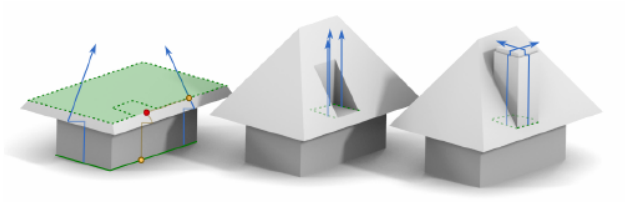


Fig. 6. Left: The plan (solid green line) and profiles (blue lines) define the shape of the structure. The anchors (orange) locate the chimney (red). A natural step is inserted into the building at the anchored location (dashed green lines). Middle: The finished 3d geometry, showing the profiles for the new edges. Right: Alternative natural step which adds an additional rectangle into the plan (dashed green lines) to specify a chimney.

for example a line of dormer windows on a roof. Anchors may also be used to specify the location of complex external features, such as windows and doors, described by arbitrary meshes.

Faces of the output model can be identified by adding *tags* to the profile segment. These are represented by small triangles in the user interface. Once the manifold is complete, the faces that were generated from the specified profile segment are post-processed in a particular way, for example to add tiles to the roof.

3.7 Modeling Larger Environments

We provide tools to model larger environments by example. We implemented several feature extraction algorithms to automatically label the edges of a building's plan. Example labels are length $\in \{short, medium, long\}$ and orientation $\in \{street, side, back\}$. The most important label is an angle computed by orienting the building to the street and mapping the normal vectors of the footprint edges to the unit disk. We assume that each footprint in the environment is labeled with a building type by another procedural algorithm. For each building type we can assign one or multiple profiles to each edge type including a probability value if more than one profile is assigned.

4. COMPUTING PROCEDURAL EXTRUSIONS

In this section we give an overview of the procedural extrusion algorithm. We begin by defining the terms used in the algorithm, the inputs and outputs, before outlining the many possible events that take place. Finally we give details for the computation of each event type.

4.1 Definitions

In this paper we compute an architectural shell in 3d Euclidean space with a xyz world coordinate system. The up direction is along the z axis. See Fig. 7 for an illustration of the terms.

A *(floor) plan* is a planar partition (a straight line planar embedding of a planar graph) that divides a plane into *inside* and *outside* regions. A plan has corners and edges. A plan is embedded in a plane parallel to the xy-plane (the ground plane), so that all corners of a plan have the same z (height) value. We require that the boundaries of a plan are a non-intersecting collection of oriented polygons. The inside is on the left-hand side of each oriented polygon edge.

The polygons are typically oriented counter-clockwise, but polygons describing holes are oriented clockwise. Additional bounded regions may be recursively located inside a hole. The jth polygon is described by n^j polygon corners $c_i^j \in R^3$ with $1 \leq i \leq n^j$. Each corner c_i^j is connected to the next corner (according to the polygon orientation) by an *edge*, e_i^j. In everything that follows, indices should be treated cyclically, so that in a polygon with corners c_1^j, c_2^j, and c_3^j, the corner c_4^j means c_1^j.

Each edge in a plan is associated with a *direction plane*, dp_i^j, which contains the edge. It is defined by an angle θ such that $-\pi/2 \leq \theta \leq \pi/2$. A vertical direction plane has $\theta = 0$, whilst a direction plane oriented towards the inside (outside) satisfies $\theta > 0$ ($\theta < 0$ respectively). The angle is measured between the direction plane and a vertical plane that also contains the edge.

A *profile* is a polyline that is used to control the direction plane of an edge. A profile is modeled in a local 2d wz-coordinate system and consists of a list of m points t_i. The location of point i is $(t_i.w, t_i.z)$ and we require that $t_i.z \leq t_{i+1}.z$. The profile defines $m-1$ angles, $\theta_1..\theta_{m-1}$. The angle θ_i is calculated as the clockwise angle between a vertical line and the line t_i to t_{i+1}. The angle lies in the range $-\pi/2 \leq \theta_i \leq \pi/2$ and the final angle is constrained such that $\theta_{m-1} > 0$.

4.2 Overview

We describe the input, the output, and give an outline of the algorithm.

Input: The input of the algorithm is a (floor) plan, called the *input plan*, profiles associated with the edges of the input plan, profile offset events, and anchor events. Anchor events specify the location of plan edits or a mesh instance.

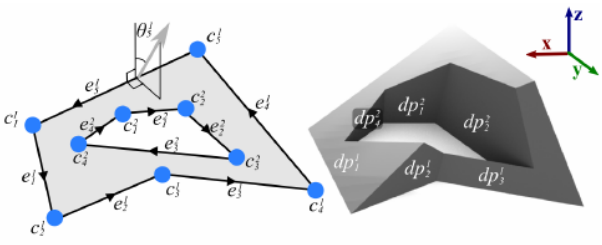


Fig. 7. Our algorithm constructs the architectural shell, shown on the right, for an input plan, shown on the left. In this simple example, each profile only has a single segment; Adding additional segments to the profile eventually allows us to model an entire building, including the walls. The input is defined by the corner positions c_i^j, the angles θ_i^j, and the corner connectivity. The output is a shell consisting of faces on the respective direction planes, dp_i^j.

Output: The main output of the algorithm is an *architectural shell* (3d mesh) in the xyz world coordinate system. In the non-degenerate case the shell is watertight and two-manifold. An architectural shell is a polygonal mesh stored in a half-edge data structure. For the sake of clarity we refer to these output edges as *arcs* (after Aichholzer et al. [1995]). The half-edge data structure stores a set of vertices in R^3, a set of arcs between the vertices, and a set of planar faces which may contain holes. Faces are defined by a counter-clockwise ordering of arcs.

The architectural shell can then be post-processed to apply textures, add procedural geometry (such as roof tiles), and attach meshes at anchor points.

```
main begin
    Q = new priority queue;
    foreach corner c_i^j ∈ inputPlan do
        foreach plan edge e_i^j ∈ planDataStructure do
            p1 = e_i^j.directionPlane;
            p2 = c_i^j.previousEdge.directionPlane;
            p3 = c_i^j.nextEdge.directionPlane;
            IE = intersect (p1, p2, p3);
            /* Queue ordered by z-height                    */
            Q.insert (IE, IE_i.z);
    /* Insert edge direction events, profile offset
       events and plan events into the queue                */
    foreach event ue ∈ userEdits do
        Q.insert(ue, ue.z);
    sweepZ = 0;
    while ! Q.empty() do
        event = Q.nextEvent();
        if event.position.z ≥ sweepZ then
            sweepZ = event.position.z;
            /* handleEvents may insert additional
               events into the queue                        */
            handleEvent(event);
end
```

Fig. 8. Pseudo-code for the main loop.

Outline: The algorithm extrudes the input plan using a sweep plane algorithm. At each height of the sweep plane a 2d cross-section through the building is another 2d plan. We call the plan associated with the current sweep plane the *active plan*. To extrude a plan, each plan edge moves to be colinear with the intersection of the direction and sweep planes. This movement and the implicitly defined geometry is straightforward until an *event* occurs. During events, we process modifications to the active plan. After inserting edges into the active plan, we must recalculate the intersection events between the direction planes. The core algorithm, Fig. 8, is a loop that handles events according to their height.

Data structures: The *plan data structure* describes the implicit active plan on the sweep plane [Felkel and Obdrzalek 1998]. This structure is a doubly linked list of corners. Each corner has a pointer to the next corner and the previous corner (assuming counter-clockwise order) and a pointer to its previous and next edges, Fig. 10. At the beginning of the algorithm the plan data structure encodes the input plan. During the sweep the data structure is updated to encode any changes to the active plan. To give a concise description we define the algorithm by discussing changes to the implicit active plan.

The second important data structure is a priority queue that sorts events by ascending height. Intersection events are automatically created, while others (edge direction events, profile offset events and anchor events) are defined by the user. We fill the priority queue with a large number of potential intersection events. An intersection event occurs wherever three or more direction planes intersect.

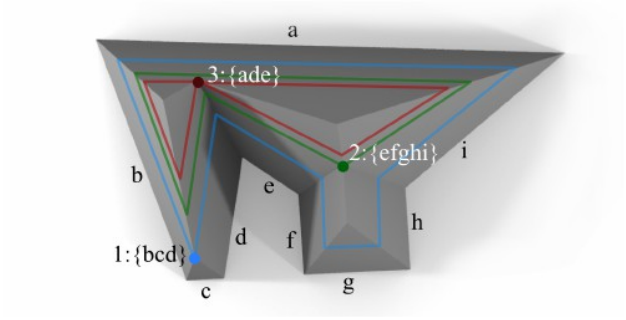


Fig. 9. An example construction demonstrating basic intersection events, and the active plan (blue, green and red lines) on the sweep plane after each event is processed. In (1) three adjacent direction planes collide at an *edge event*. In (2) we see a vertex event where more than three direction planes collide at one point. Finally, in (3) we show a *split event* that splits the area bounded by the plan.

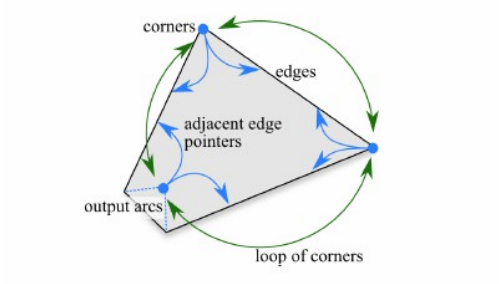


Fig. 10. The plan data structure, shown part way through the sweep.

Given the active plan at all event heights, the generation of the half-edge data structure describing the architectural shell and subsequent triangulation of shell-faces is fairly straightforward.

4.3 Description of Events

In this section we describe the events encountered as the sweep plane ascends.

Generalized Intersection Event: There are three event types, given by previous authors [Felkel and Obdrzalek 1998; Eppstein and Erickson 1998], which automatically occur to the edges in the active plan as the sweep plane rises. *Edge events* occur as the length of an edge shrinks to zero. When an edge shrinks to zero the direction planes defined by three consecutive (linked by corners) edges collide (Fig. 9, 1). *Split events* take place when two adjacent direction planes, and one non adjacent direction plane collide (Fig. 9, 3). These split the region bounded by the active plan into two parts. Finally *vertex events* occur in the degenerate case when more than three direction planes collide at one point (Fig. 9, 2).

Unfortunately, we did not find this categorization of events helpful to designing an algorithm. In practice architectural models give rise to a large number of degenerate events and the implementation is dominated by special event handling. Since edge and split events are special cases of a vertex event, we introduce one *general intersection event* that consists of an arbitrary number of direction planes, bounding one region, intersecting at one point. See

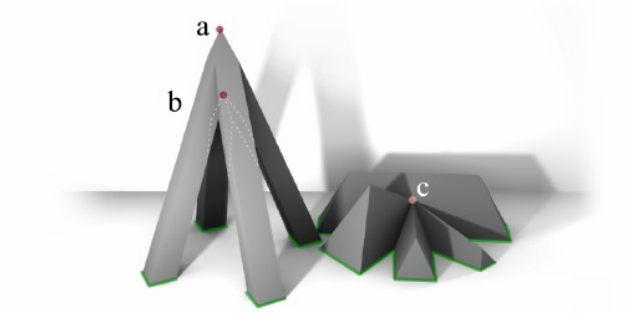


Fig. 11. Procedural extrusions that give rise to three degenerate cases. At a and b, four faces collide at one point. Point c shows seven faces colliding from a variety of angles, including horizontally.

Fig. 11 and 12 for four examples. We introduce a new algorithm to resolve this generalized intersection event that uses *chains* of edges involved in the intersection.

Edge Direction Events: An edge direction event occurs when a profile curve changes direction. The event updates the angle and direction plane associated with a set of edges in the active plan.

Profile Offset Events: Profile offset events occur at heights specified by user edits. Intuitively, a profile offset event results in additional inside regions being added to the active plan at the specified height.

Anchor Events: Anchor events specify locations on the architectural shell, and are defined by the user. There are two types of anchor events. *Plan Edit Anchors:* These modify the active plan to insert new features such as chimneys, or dormer windows. *Mesh Anchors:* These store the location of the anchors as an attachment point for geometry.

4.4 Generalized Intersection Event

Generalized intersection events perform topological changes on the active plan to ensure that it never self-intersects as the sweep plane ascends. These events are automatic, not user driven.

There are many possible topologies that can give rise to a generalized intersection event. Previous authors have described how to adjust the active plan to deal with split and edge events. These are the most frequent events when the input is a random polygon. We describe a generalization of these techniques to deal with the most likely class of topologies when dealing with architecture, a *locally connected region*. When our interface is used to model architecture, these account for the vast majority of events. A locally connected region is a region, that immediately before the event is locally equivalent to a topological disc, Fig. 11 (abc). In a single event the locally connected region may be either an “inside” or “outside”.

In addition to locally connected regions, there are several unlikely classes of increasingly degenerate events in which the intersecting edges define a nested boundary, Fig 15. When this situation occurs we give a warning in the user interface that the output may be undesirable.

Event Detection: We use expanded bounds for intersection location clustering. This addresses two stability problems.

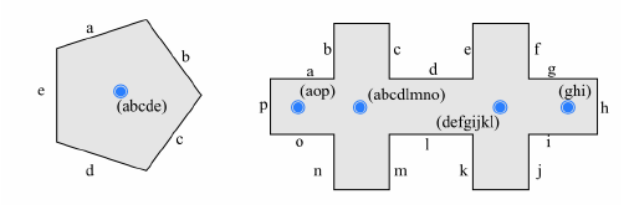


Fig. 12. Left: Five faces creating an intersection event. Right: Events can interfere with each other if they have the same height, in this case the four points share a roof ridge.

First, in symmetrical inputs, like architectural plans, it is very common for more than three direction planes to meet at a point. To avoid degenerate output in a floating point situation it is necessary to identify intersections whose locations are close together, and treat these as a single event. See Fig. 12 (left) for an example.

Second, direction plane intersections that are far apart from each other can interfere if they are close to one other in height, Fig. 12 (right). It is necessary to detect and handle these together to ensure the region bounded on the sweep plane does not self-intersect and to resolve the ambiguities that can occur (described in Sec. 4.8).

Event Clustering: After initialization we iteratively process (potential) events stored in the priority queue. To address the two previously mentioned event detection problems, we cluster the events in two directions. We poll the priority queue to collect all intersection events whose height, z, is within some threshold, δ_1, of the initial event. Second, we cluster all the events according to their location after projection onto the xy sweep plane. The clustered volume is therefore a cylinder of radius δ_2 and height δ_1. See Fig. 13 for an illustration of the clustering step. For our building floorplans with a size in meters we use double precision floating point representations and values $\delta_1 = 10^{-4}, \delta_2 = 10^{-6}$, found through trial and error on our large procedural floorplan set. There are certain pathological inputs which cause this clustering stage to fail. An example would be a row of events, each within δ_1 of another, which could contain an arbitrary number of events. In such a case we alert the user with a warning message, but none of the users reported such a situation.

Input: The input of a generalized intersection event is a point $l \in R^3$, and a set of three or more active plan edges, f, whose associated direction planes intersect at l. The point l is calculated as the center of the clustered volume.

Output: The output of a generalized intersection event is an updated active plan. This represents the bounded region on the sweep plane after the event.

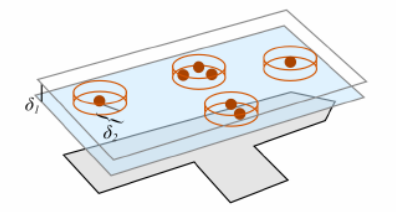


Fig. 13. When an event is processed we simultaneously extract all intersection events within a height of δ_1. Then we cluster all events that are within a cylinder of radius δ_2 and height δ_1.

Chain construction: We process the edges involved in the clustered intersection events into a set of *chains*. A chain defines a connected portion of the active plan boundary involved in the event, Fig 14 (a). A chain, h^i, is a list of consecutive active plan edges, $\epsilon^i_1 ... \epsilon^i_{hmax_i}$. A cyclic *chain list*, b, contains all such chains, $h^1 ... h^{bmax}$ (we assume a cyclic index). The list is ordered by the edge's orientation around l.

The list of chains, b, is now processed to update the active plan in two stages. First within each chain (intra-chain), and then between the chains themselves (inter-chain).

Intra-chain handling: In a chain of 2 or more edges, the interior edges shrink to length 0 as we approach the intersection event, Fig 14 (ϵ^2_2). Therefore in the intra-chain stage we remove all interior edges from a chain h^i, leaving only the start, ϵ^i_1, and the end, $\epsilon^i_{hmax_i}$, of the chain as shown in Fig. 14 (cd). That is, if $hmax_i \geq 3$, then edges $\epsilon^i_2 .. \epsilon^i_{hmax_i - 1}$ are removed from the active plan, being replaced by a new corner at l, connecting the end of ϵ^i_1 to the start of $\epsilon^i_{hmax_i}$.

Inter-chain handling: In a typical intersection event, the closest edges in adjacent chains move into each other. To allow this without self-intersections the inter-chain stage takes place between each adjacent pair of chains, h_x and h_{x+1} in the cyclic chain list b. Firstly, if any chains contain only one edge, we split that edge by inserting a corner at l, Fig. 14 (de). Secondly, for each pair of adjacent chains we create a new corner at l and connect the start of the last edge in the proceeding chain, $\epsilon^x_{hmax_x}$, and the end of the first edge in the following chain, ϵ^{x+1}_1, Fig. 14 (e). Finally the inter-chain stage finishes by removing any unreferenced corners from the active plan.

In addition to this basic technique, there a several implementation issues that we address — the filtering of invalid events, checking for chain intersections and local non connected events.

Filtering invalid events: Before the clustering stage we remove any invalid edges from the edge set, f. Because the intersections are detected using unbounded direction planes, there may be edges in f that do not approach l on the active plan. Such edges are removed from f. The line defined by the intersection of the direction plane and the sweep plane may pass close to l, however the line-segment defined by the associated active plan edge may not. A small epsilon range, δ_3, expands the length of the edge and ensures that collisions occur reliably. On our inputs we find $\delta_3 = 10^{-5}$ a sufficient margin.

Second, an edge may have been removed from the active plan by a previous event. These edges are also removed from f. After filtering, if the number of edges in f is less than 3, the event is ignored.

Post inter-chain intersections: There are rare situations where the chains after the inter-chain stage no longer form a valid plan on the sweep plane. To test if the chains form a valid plan we predict the chain locations on a plane higher than the active plan. If any of the chains intersect each other we use an application of the winding rule to calculate valid region boundaries for the current active plan. This may re-orient some edges, as well as insert new edges or corners into the active plan.

Local non connected events: The above handling of locally connected events is sufficient to create large cityscapes, Fig. 2. There are however, a class of degenerate topologies that can occur on the active plan, which are not connected, such as in Fig. 15. While we do not present a solution for each of these classes, the following observations are made.

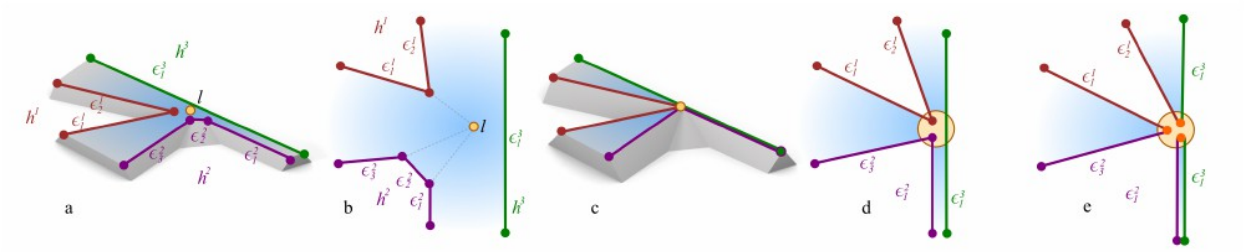


Fig. 14. Active plan modification during a generalized intersection event. a) The active plan just before the intersection of chains h^1 (red), h^2 (purple), and h^3 (green) at the point l. The chains consist of edges, ϵ. b) The topology just before the event. c) The active plan geometry at the event, note the disappearing region bounded by coincident edges ϵ_1^2 and ϵ_1^3. d) The topology of the chains after the intra-chain stage. An edge, ϵ_2^2, has been removed. e) The topology of the active plan after the inter-chain stage. The edge in a chain of length one, ϵ_1^3, has been divided at l. After the edges are linked at l, the active plan has been split into three regions. The three new corners (e, orange) are at the same location, l, but they are expected to move in different directions over the course of the extrusion process.

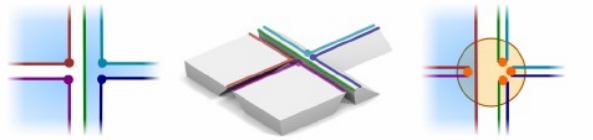


Fig. 15. A complex, non-connected region, causing an intersection event. Left: The active plan just prior to the intersection event between all the 9 edges. Middle: The output shell at this height, showing the edge angles and the colliding edges. Right: One of several non-symmetrical solutions that removes all but one edge.

If the edges in a chain form a closed loop, the chain may be simply removed. If there is more than one chain of length one, the associated chain edges must be parallel and the geometry between such adjacent chains may also be removed. Finally, if a chain is nested inside another chain there are situations where inter-chain updates no longer work. Here we just note that reversing a section of the enclosing chain is enough to keep the area enclosed on the active plane well formed. We note that we could not find an example where such degenerate events were part of a meaningful architectural construction.

4.5 Edge Direction Events

A set of edge direction events are created for each profile. An edge direction event updates the angle and direction planes of a set of edges. There are two types of edge direction event, *standard* and *near horizontal*. Standard edge direction events are constructed from a single angle in the plan, while a near horizontal edge direction event is constructed from two consecutive angles and a distance. These values are calculated from the profile polyline.

4.5.1 *Standard Edge Direction Events.* **Input:** A set of edges, f, in the active plan, each associated with the same profile and a single new angle for all the edges, γ. **Output:** A new active plan which replaces the original.

For each of these edges $e_i^j \in f$, we update the associated direction plane by setting its angle to γ. The edge, e_i^j, continues to propagate over the sweep plane as defined by the new angle.

4.5.2 *Near Horizontal Edge Direction Events.* We need a separate approach as the angle associated with an edge, θ, approaches $\pm\pi/2$, as two parallel (horizontal) direction planes do not intersect to form a line. Additionally, as the angle approaches these limits we are colliding near coplanar planes, causing numerical instability. As Fig. 16 illustrates, we first increase the angles for numerical robustness, recursively apply procedural extrusions, and then project onto the sweep plane. This produces the required horizontal surface.

Input: A set of edges in the active plan, f, associated with the profile, a distance, d, a direction angle, γ, and a following angle, ζ. The angle $\gamma \approx \pi/2$ ($\gamma \approx -\pi/2$) specifies the direction of the horizontal as towards the inside (respectively outside) of the active plan. ζ specifies the angle of the following non-horizontal edge event. **Output:** A new active plan which replaces the original.

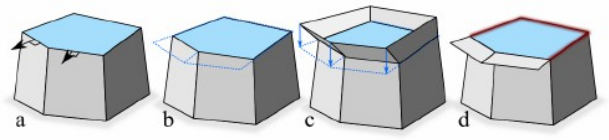


Fig. 16. The horizontal section desired (b) can be created by an additional application of procedural extrusions to calculate the offset in the given direction. After flattening (c) unchanged edges (red, d) are ignored.

First we create a temporary plan as a copy of the active plan. For each edge in the original plan, e_i^j, and associated angle θ_i^j, the temporary plan has an edge E_i^j, and associated angle Θ_i^j. Secondly we update the angles in the temporary plan according to the following mapping:

$$\Theta_i^j = \begin{cases} \tan^{-1}(d) & \text{if } e_i^j \in f \text{ and } \gamma > 0 \\ -\tan^{-1}(d) & \text{if } e_i^j \in f \text{ and } \gamma < 0 \\ 0 & \text{otherwise} \end{cases}$$

A recursive application of procedural extrusions extrudes the temporary plan for a height of one unit. The temporary active plan is projected onto, and replaces, the active plan in the original procedural extrusion instance. That is, e_i^j is replaced by E_i^j if it exists in the updated plan. If it does not exist in the updated plan e_i^j is removed

from the active plan. The location of E_i^j is projected onto the original active plan. Finally the values of θ in the original skeleton are updated using the mapping:

$$\theta_i^j = \begin{cases} \zeta & \text{if } e_i^j \in f \\ \theta_i^j & \text{otherwise} \end{cases}$$

Occasionally multiple edge direction events occur at the same height. In this situation the direction events are sequenced by the order of user creation. The user can manually override this priority.

4.6 Profile Offset Events

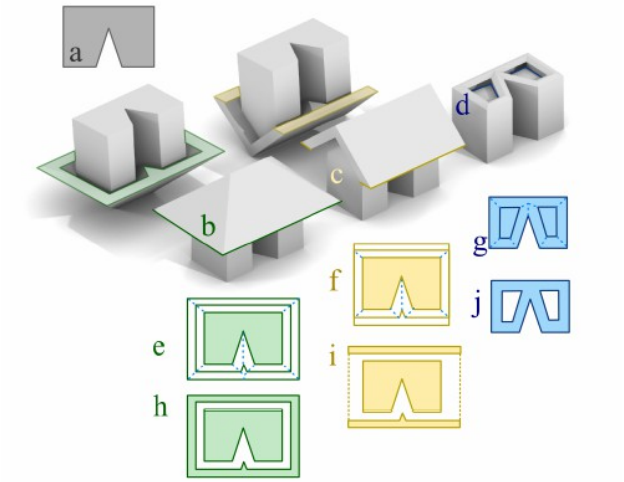


Fig. 17. Some meshes that can be computed from an input plan (a) using profile offset events. Buildings b and c are shown in two orientations. By creating two offset boundaries (e) that define an offset region (h), an overhanging roof (b) can be generated from an arbitrary plan (a). If two edges are disabled in the profile offset event, open-ended roofs can be created (c,f,i). Finally, by offsetting inside the active plan, walled roofs can be created (d,g,j).

Profile offset events specify the start of overhangs. The difficulty of specifying and handling profile offset events comes from the procedural definition. While it is easy to specify overhangs for a given region, the geometry must adjust itself according to subsequent user edits. Our technique must procedurally perform changes to the active plan without creating awkward self-intersections.

At a profile offset event an additional inside region, called an *offset region*, is inserted into the active plan (see Fig. 17). Two offset boundaries are grown from the active plan to enclose the new offset region. We introduce new edges and corners into the active plan to represent this newly enclosed region on the sweep plane. The new edges are classified as inside, outside, or side, depending if the edge stems from the first boundary, the second boundary, or an intersection operation between the two boundaries described later in this subsection.

Input: A map for each edge in the active plan, e_i^j to a tuple, $t_i^j = \{disabled_i^j, dist\_inside_i^j, dist\_outside_i^j, profile\_inside_i^j, profile\_outside_i^j\}$ and a single $profile\_side$. The variable

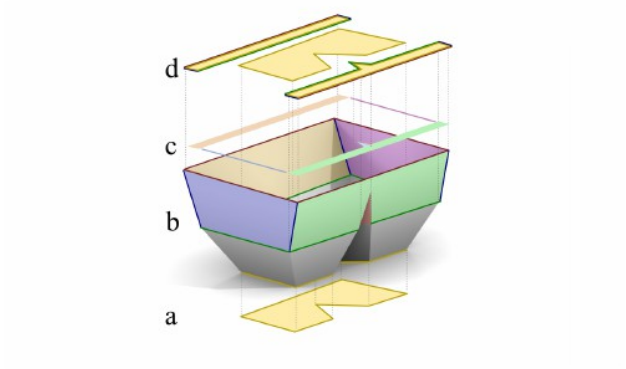


Fig. 18. The recursive application of procedural extrusions (b) to a plan (a) from Fig. 17 (c). The faces between $z = 1$ and $z = 2$ are projected onto the primary active plan (c), before being merged (d). Zero area faces (blue and purple) are removed, and profiles assigned based on the origin of the edge. In (d) green edges are assigned $profile\_inside$, red $profile\_outside$ and blue $profile\_side$.

$disabled_i^j$ is a boolean value that specifies if the offset region associated with this edge is present in the output; $dist\_inside_i^j$ and $dist\_outside_i^j$ are real values that define distance and direction from the active plan of the inside and outside offset boundaries; $profile\_inside_i^j$, $profile\_outside_i^j$ and $profile\_side$ are profiles. We require that all values of $dist\_inside_i^j$ and $dist\_outside_i^j$ have the same sign; a positive (negative) sign indicates an offset (respectively inset) of the active plan. To ensure proper topology on the active plan, the distance, $dist\_inside$, is constrained to be non-zero. **Output:** The output of an offset event is an updated active plan, typically with the additional region defined either inside or outside of the input active plan.

We create a temporary plan as a copy of the primary (input) active plan. For each edge in the primary plan, e_i^j, the temporary plan has an edge E_i^j, and an associated profile, $profile\_recursive_i^j$. Edge E_i^j is constructed by projecting e_i^j onto the plane $z = 0$. The profile $profile\_recursive_i^j$ defines the angles $\Theta_i^j = tan^{-1}(dist\_inside_i^j)$ at $z = 0$, and $\Theta_i^j = tan^{-1}(dist\_outside_i^j)$ at $z = 1$. We execute a recursive application of procedural extrusions using the temporary plan as input. It is executed from height 0 to 2, to create a temporary output shell. Faces of the shell between the planes $z = 1$ and $z = 2$ are projected onto the primary active plan, forming the offset region, Fig. 18.

The projection associates each tuple, t_i^j, with an offset region in the primary active plan. The entire offset region is bounded by the projected edges, r. Additionally the projection defines a 1:1 mapping between the new edges, $e_l^k \in r$, and a subset of the temporary shell's arcs A_l^k. We remove from the primary active plan any edges in r that enclose an offset region of area 0 or that are associated with a tuple containing a value of $disabled_i^j = true$. We update the profile, $profile_i^j$, associated with each edge, e_i^j, in the primary active plan according to the function:

$$profile_i^j = \begin{cases} profile_i^j & \text{if } e_i^j \notin r \\ profile\_inside_i^j & \text{if } A_i^j \text{ lies in the plane } z = 1 \\ profile\_outside_i^j & \text{if } A_i^j \text{ lies in the plane } z = 2 \\ profile\_side & \text{otherwise} \end{cases}$$

Finally we merge adjacent parts of the offset region to avoid self-intersections. We remove the corresponding edges and corners from the active plan.

4.7 Insertions into the polygon

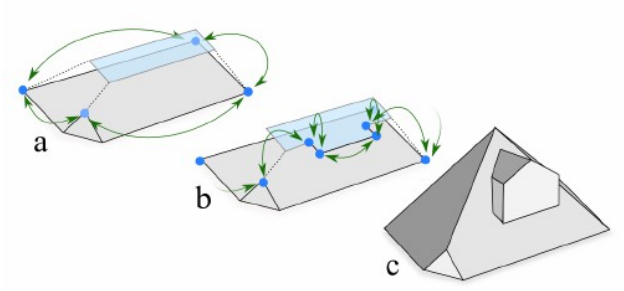


Fig. 19. Inserting a plan edit into the active plan during execution. a) The plan data structure (blue dots, green arrows) implicitly defines the active plan (cyan). b) To insert new edges into the active plan, corresponding edges are linked into the plan data structure. c) The resulting architectural shell.

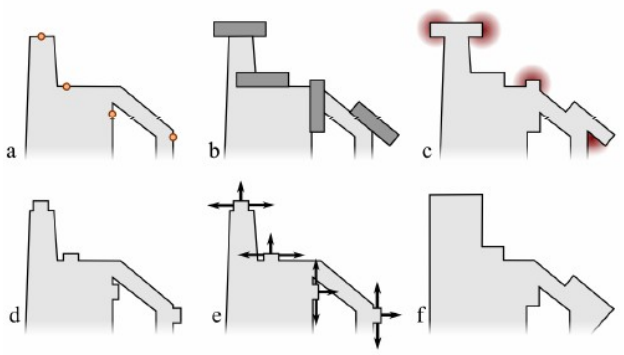


Fig. 20. Given an intricate plan, calculating a robust perturbation is challenging. Forced steps are positioned at the location of the anchors (a, orange). These are combined with the boundary using a geometrical union operation. However many geometry artifacts are undesirable (c, red) in an architectural situation. Given natural steps at certain positions (a, orange), small changes to the boundary are made (d), which are then grown (e) using a recursive application of procedural extrusions, to create more natural geometry (f).

Plan edits introduce discrete changes to the active plan at specified heights. We describe how plan edits operate efficiently and detail two methods to define them.

When performing a plan edit, some edges are deleted, some edges are moved, and some edges are inserted, Fig. 19. These new edges are at the height of the current sweep plane.

Our user interface offers two types of plan edits. Inserting an arbitrary shape gives the largest variety of geometric designs. However these *forced steps* offer no guarantees that the resulting active plan will not self intersect and create an invalid topology. The challenge comes again from the procedural nature of our approach and the fact that the edit has to work for all input plans. *Natural steps* offer a solution to this problem by using a recursive application of procedural extrusions to insert edges into the active plan.

Natural steps are calculated on the active plan at a given height by amending a small (typically 10^{-3} by 10^{-3}) protrusion. This is offset by a recursive application of procedural extrusions such that it does self intersect, Fig. 20. This is similar to the edge direction events of Sec. 4.5. This application of procedural extrusions is constructed by assigning $\theta = 0$ to all edges not part of the feature, and a user defined θ to those edges in the protrusion. The resulting temporary active plan is calculated at a specific height, and this is incorporated into the original active plan. The new edges in the active plan have the relevant profiles assigned to them.

4.8 Ambiguities in Procedural Extrusions

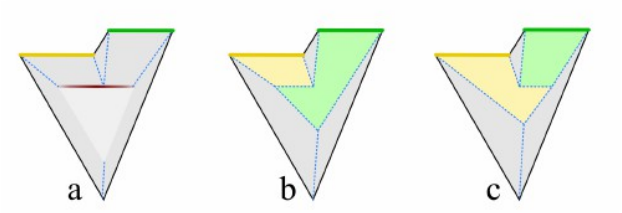


Fig. 21. An ambiguous situation that arises in the case of a concave input plan, a. The intersection of the yellow and green edge's direction planes gives two possible output arc directions (a, red). This may be resolved into two ways, b,c.

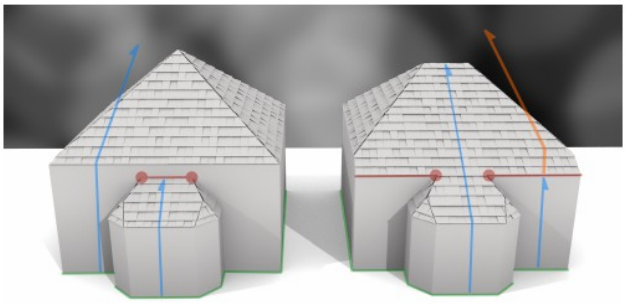


Fig. 22. Two identical bay windows that lead to the same two events (red circles) involved in an ambiguous situation (red line). To resolve the ambiguous situation, a single edge must be chosen to replace the others. The building on the left (right) resolves the ambiguity using the volume maximizing (respectively minimizing) priority technique. The resulting unused section of the original profile is shown in orange. Note that in each case, two ambiguous events occur at the same height, and must create globally consistent output.

We show that procedural extrusions (as well as the weighted straight skeleton [Eppstein and Erickson 1998]) are ambiguous in the concave case. Different modeling choices lead to different ambiguous-case resolution strategies, Fig. 22.

The ambiguous case may arise when two (or more) neighboring edges in the active plan become colinear on the same side of a region. This happens, for example, when edges previously separating

the colinear edges are eliminated. We may also arrive in this situation if the input, or any of the plan edits, introduce colinear edges.

If the two neighboring and colinear edges bound different sides of a region the output is an arc representing a ridge and the computation proceeds as normal, assured that at the opposite end of the ridge the two edges will collide again [Felkel and Obdrzalek 1998]. This is not an ambiguous event and we can distinguish the regular roof ridge case from an ambiguous event by making use of the fact that we use oriented edges to describe a plan. When the edges have the same orientation (they bound the same side of a region) we are not able to determine the direction of the output arc, Fig. 21. This produces an ambiguity.

The individual ambiguous events need to be solved consistently from a global perspective, Fig. 21; This is one reason for the vertical clustering outlined earlier. All colinear consecutive edges involved in an intersection on the active plan are grouped together. We resolve the situation by merging all consecutive edges into one and applying the profile of the edge that has the highest priority. Then we remove the other edges involved in the ambiguous events. To select the one edge we assign a priority ordering over the edges and choose the edge with the highest priority.

We introduce three possible priority schemes. It is interesting to note that most architectural roof structures (such as bay or dormer windows) enclose the maximum volume in the ambiguous case. This leads to our default scheme in which the highest priority edge, e_i^j, has the lowest (closest to $-\pi/2$) associated angle, θ. Alternately the minimum case (largest associated θ) may be useful when estimating conservative offsets. The third option is to manually define the priority function in the user interface. Section 6.2 describes situations where it is desirable for the user to manually define the priority function.

5. RESULTS

5.1 Modeling Results

In this section we show several interesting applications of modeling with procedural extrusions.

Fig. 23 shows many typical architectural shells that are not possible using just the straight skeleton or other existing procedural modeling tools. We can also create buildings with horizontal roof overhangs, such as Fig. 24. The alcoves and columns show how disconnected regions can merge together and interact. This is only possible because we allow negative angles for the roof planes.

Procedural extrusions may be used on a large scale to describe cityscapes. We created a procedural model using about 6000 footprints from Atlanta (see Fig. 2). The current model has three million polygons, 5 different building styles, took 20 minutes modeling time, 10 minutes to compute the procedural extrusions, and 15 minutes to render. Our current limitation is that we were not able to find a rendering infrastructure to render a few hundred million polygons of a detailed model. We therefore had to omit ornaments and some details of the roof constructions from the designs.

We implemented the proposed system in Java and measured the running time of our system on 64bit 2.6GHz Xeon.

We created a procedural model for town homes adjacent to a curved street, Fig. 26. The street can be reshaped interactively, while the building models adapt to the new footprints. Finally, procedural extrusions can be used to model other architectural features such as windows or moldings, Fig. 27.

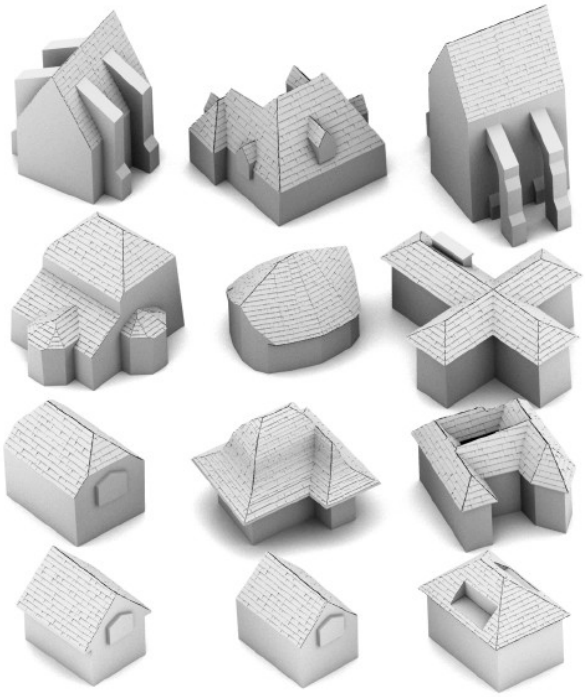


Fig. 23. From top, left: buttress, dormer windows, flying buttress, bay windows, curved plan, eight faces meeting on a symmetrical footprint with a chimney, hipped roof, curved roof, a horizontal overhang, an overhanging gable, standard gable and interior dormer windows

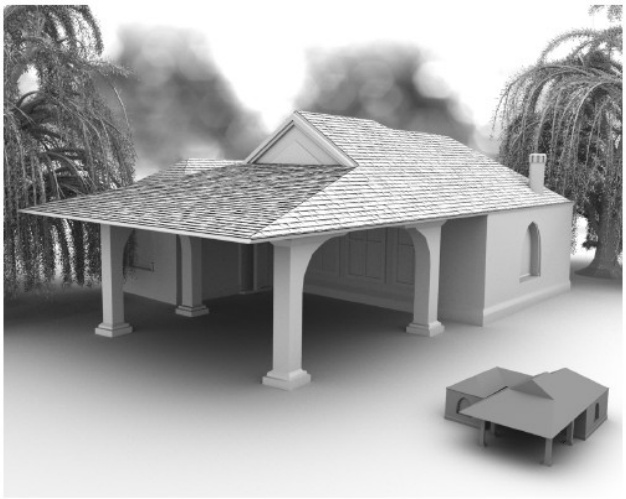


Fig. 24. Inset: the output of our procedural extrusions using a complex footprint, horizontal sections and plan edits. We are able to create pillars, covered parking and alcoves respectively. Main: A procedural condo with roof texture surrounded by procedural trees

Fig. 25. The example cases and modeling statistics. *v* Vertices in modeled plan (additional vertices); *l* Polygons in modeled plan (polygons in library plan); *p* Number of profile sections in model; *s* Number of natural steps designed (number of natural step applications); *o* Number of offset events.

6. EVALUATION

To evaluate the skeleton as a modeling primitive we constructed 50 buildings. Here we detail the process we undertook to perform the modeling.

6.1 Evaluation Setup

We modeled each building from a plan and a perspective image. A set of four simple meshes (Fig. 28) were used to add detail to

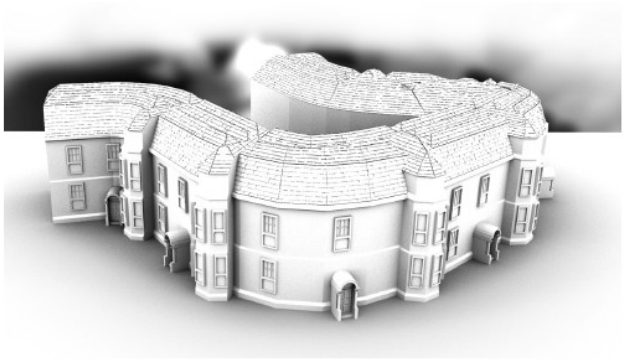


Fig. 26. A procedural model that renders a street from a spline. In this case the street was generated by four points defining the street's curve. Seed points were grown using another application of the skeleton to create the building footprints.

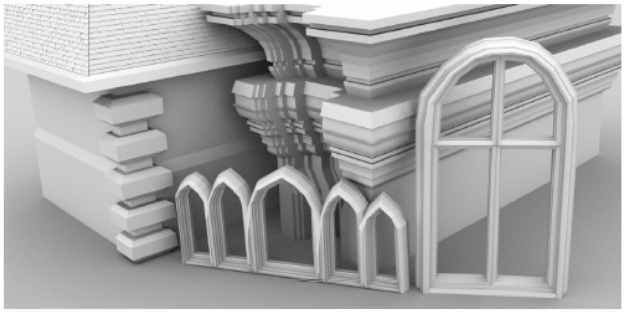


Fig. 27. Using a creative set of profiles, a wide range of architectural features can be created. By setting the input in a different plane, various windows may be extruded.

the structures. The events used for modeling were edge direction events, profile offset events and natural steps.

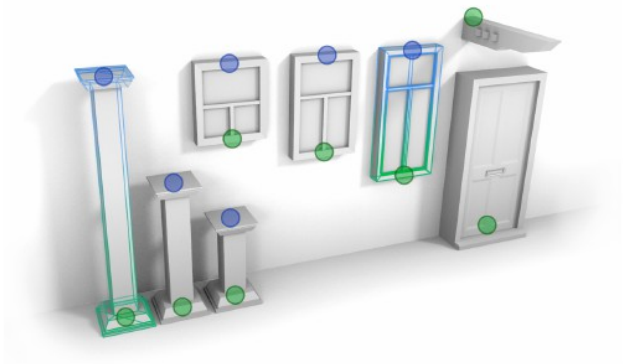


Fig. 28. The four example meshes used in the evaluation. The meshes are parameterized via control points (blue and green circles) and can be instanced to different sizes.

We undertook the evaluation with the goal that all major roof features from the elevation drawings should be present, although smaller details (such as cornices, plumbing and decorative windows) may be excluded. We traced the plans from those specified or aerial views of the property. The construction of profiles and positioning of features was performed by eye by the first author of this paper.

The first 45 buildings were taken from a library of ready designed architectural styles for family homes [Hanley Wood, LLC. 2010]. We modeled the first example in each of the categories the library provided. The library contained styles as diverse as *ranch* or *Dutch* (Fig. 25, examples 13 and 32 respectively), however much of the stylistic content was dependent on architectural details that were replaced with our simple meshes. Because the plans were predesigned, they had predominantly 90° and 45° degree angles between floorplan edges. That is, the design was not constrained by environmental features. To provide more challenging examples, we chose an additional five buildings from European cities that had irregular plans (Fig. 25, examples 46-50). These buildings were modeled from satellite and aerial views, Fig. 29.

The modeling times ranged from 20 to 120 minutes with a mean time of 63 minutes. Features on the input plan smaller than 30cm were not modeled. We also recorded a number of additional metrics for each building: the number of vertices in the input plan and in the model; the number of corner-loops in the input and in the model; the number of profiles in the model, the number of offset events, the number of natural step templates and the number of instances of those steps.

6.2 Evaluation Results

It was possible to model all the buildings using our interface. Some roof-lines were easier than others, and in this section we describe some of the problems encountered.

The most common issue when modeling was the construction of roof areas that contained edges not specified in the input plan (Fig. 30 (a). In these circumstances it was necessary to add extra edges to model these features. These would either be added in the plan, leading to the difference between the vertices in the input plans and the model in several of the examples, or by natural steps at certain heights.

We share a limitation with the straight skeleton that certain smaller edits to the footprint can result in bigger changes to the roof surface [Eppstein and Erickson 1998]. For example when two adjacent edges with different angles are nearly parallel, the behavior of the resulting roof can be erratic as the angle between the edges is set to greater than, or less than zero. In practice these edges do not appear often in architecture, and we often end up adding a perpendicular edge (Fig. 30, a).

In several circumstances one face relies upon another, spatially separated, face to halt its propagation at the correct time, that is an edge is fated to meet another (Fig. 30, b). When another feature blocks, or changes the course of one of these faces, the other may not terminate, or collide in an unexpected location. These fated edges lead to potentially undesirable intermediate outputs while editing.

Modeling circular arches was difficult because any adjustment in the width of the arch, would have to be accompanied by a re-scaling of the profiles. Modeling techniques such as shape grammars are able to retain such semantic information to automate such a process, and it is possible to imagine a similar system for the procedural extrusions.

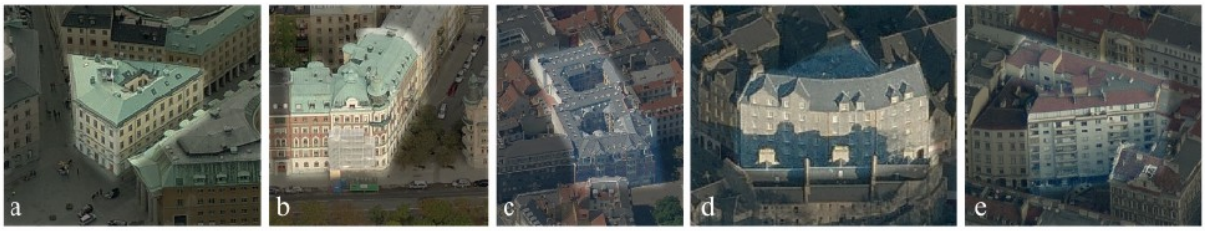


Fig. 29. Sample aerial photographs of buildings used for modeling examples 46 to 50 in Fig. 28. a,b) Stockholm, c) Copenhagen, d) Edinburgh, e) Vienna. ©2011 Microsoft Bing Maps [Microsoft Corp.].

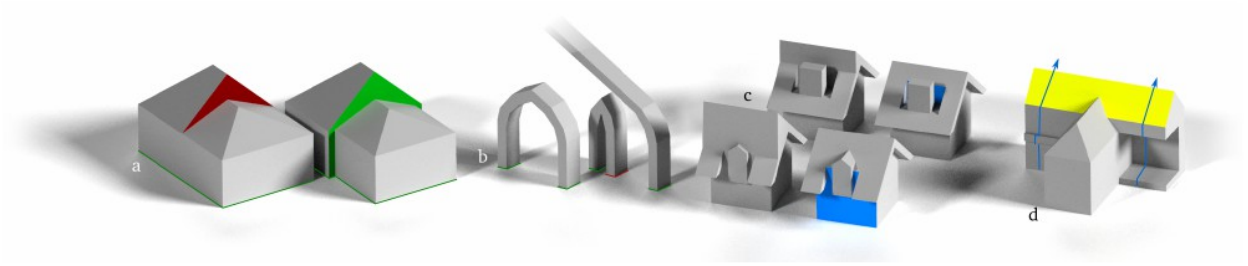


Fig. 30. a) The red roof face is not described in the input polygon(left). By creating a small change to the input polygon we can create the desired face (green). b) left: edges can be expected to collide at a certain height (green polygons), right: however when these edges are involved in other events (such as those from the red polygon), there may be undesired consequences, here a non-terminating polygon. c) Some structures (such as dormer windows and chimneys) do not obey the volume-maximizing resolution to the ambiguous case, in this situation we have to lower the ambiguous case priority of some edges (blue) to get the desired result. d) A face (yellow) may be shared between two profiles (blue lines), defining co-planar profile sections requires patience on behalf of the user.

It is not convenient to model a roof that is held only by a large number of pillars, because it is not easy to model the transition from pillars to the roof. For example, pergolas (Fig. 25, example 31) contain no walls to allow the plan to generate a roof. These were not a large part of our data set, and were approximated by walled structures of similar volume.

It was occasionally necessary to override our default of a volume maximizing priority in the ambiguous case. For example, in the case of a chimney stack or a dormer window (Fig. 30, c). To do this we used tags to specify high priority and low priority profile segments. This proved simple compared to the alternative of specifying a priority for every pair of segments.

While allowing one profile to split into two is the simple case of inserting an edge with a step, allowing two profiles to merge to one is more difficult (Fig. 30, d). We see this architectural feature as two different profiles to merge at the top of a shorter roof (Fig. 25, examples 3, 20). To design a profile with a face co-planar to another is difficult, especially if the second edge starts from an edge parallel, but not colinear to the first.

Natural steps proved very versatile for inserting edges into the polygons. For example, Fig. 25 (example 34) required a new edge internal to the plan for the back-facing wall of the tower. By positioning a wide square natural step on the end of the building, it was possible to split the polygon into two. One partition became the tower, and the other the remainder of the roof structure.

From a development perspective the algorithms are difficult to implement. It is hard to give a formal guarantee that the implementation will work correctly on all inputs. This may be observed when using our user interface, as occasionally a face will not contain enough arcs to close the area. In this case the face will not be visible to the user. This may occur once in every 5 minutes of interactive editing with multiple edits per second; It is certainly possible to construct pathological input cases. In the procedural case, we visually identified missing faces in two of the meshes, from the 6000 floorplans in the GIS database, Fig. 31.

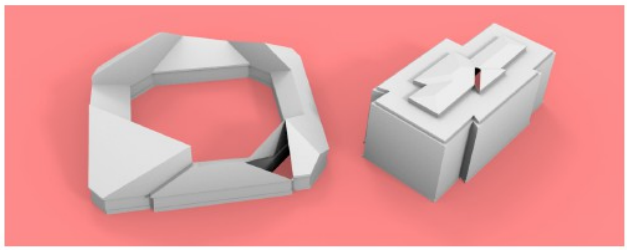


Fig. 31. The two observed examples of missing geometry. Note the missing roof sections in both buildings.

However, our modeling system is more specialized than most commercial polygonal modeling packages. The virtual model of Atlanta is unique and we argue that no existing approach can model a city of comparable (roof) complexity in reasonable time.

7. DISCUSSION

A major design decision for our system was to choose between a rational arithmetic or a floating point implementation. Our floating point implementation is better suited to interactive modeling

applications because it prioritizes interactive update speeds over high precision. A rational arithmetic approach may be important to give theoretical guarantees and such an alternative implementation would be very valuable.

Fig. 32. Left: Straight skeleton; Middle: Straight Skeleton with angle changes; Right: Procedural extrusions

We are the first to introduce an algorithm for extrusions using edges with independent per-edge angles (weights). This results in a 3d instead of a 2d algorithm. We are also the first to recognise the difficulty of independent per-edge angles. The possibility of a 2d weighted skeleton is discussed in previous work [Eppstein and Erickson 1998; Barequet et al. 2008], but no algorithm is given and the ambiguous cases were not discovered. Even though our work is based on previous work in the unweighted case, e.g. [Cacciola 2004; Felkel and Obdrzalek 1998], our modifications result in substantial improvements in the range of forms that can be produced, Fig. 32.

In previous work [Havemann 2005] the direction of the extrudes is monotonic in the upwards direction, that is they are limited to angles above the sweep plane. By using profile offset events, we can allow non-monotonic profiles.

8. CONCLUSIONS

We believe that the combination of interactive and procedural modeling is a significant boost to artists productivity and a great complement to existing modeling tools. In some sense our work is complementary to previous work by Lipp et al. [2008]. Our approach to encoding procedural models is very different from the previous shape grammar approach [Müller et al. 2006; Lipp et al. 2008]. We believe that we are the first to provide a solution for the procedural modeling of roofs, procedural modeling from arbitrary building footprints, and other complex architectural surfaces. However, previous work is better suited for placing elements on facade planes and we see some potential in combining both approaches in future work.

The main contribution of this paper is the design of the system and the set of tool choices to enable procedural modeling of complex architectural surfaces. Procedural extrusions can model many complex architectural surfaces that could not be easily modeled with previous procedural modeling tools. Examples are curved roofs, overhanging roofs, dormer windows, interior dormer windows, roof constructions with vertical walls, buttresses, chimneys, bay windows, columns, pilasters, and alcoves.

REFERENCES

AICHHOLZER, O. AND AURENHAMMER, F. 1996. Straight skeletons for general polygonal figures in the plane. In *Computing and Combinatorics*. Springer-Verlag, 117–126.

AICHHOLZER, O., AURENHAMMER, F., ALBERTS, D., AND GAERTNER, B. 1995. A novel type of skeleton for polygons. *Journal of Universal Computer Science 12,* 12, 752–761.

ALIAGA, D. G., ROSEN, P. A., AND BEKINS, D. R. 2007. Style grammars for interactive visualization of architecture. *IEEE Transactions on Visualization and Computer Graphics 13,* 4, 786–797.

AURENHAMMER, F. 2008. Weighted skeletons and fixed-share decomposition. *Comput. Geom. Theory Appl. 40,* 2, 93–101.

AUTODESK INC. Revit™. http://www.revit.com.

BAREQUET, G., EPPSTEIN, D., GOODRICH, M. T., AND VAXMAN, A. 2008. Straight skeletons of three-dimensional polyhedra. In *ESA '08: Proceedings of the 16th annual European symposium on Algorithms*. Springer-Verlag, Berlin, Heidelberg, 148–160.

CABRAL, M., LEFEBVRE, S., DACHSBACHER, C., AND DRETTAKIS, G. 2009. Structure preserving reshape for textured architectural scenes. *Computer Graphics Forum (Proceedings of the Eurographics conference).*

CACCIOLA, F. 2004. A CGAL implementation of the straight skeleton of a simple 2d polygon with holes. In *2nd CGAL User Workshop*.

EPPSTEIN, D. AND ERICKSON, J. 1998. Raising roofs, crashing cycles, and playing pool: applications of a data structure for finding pairwise interactions. In *SCG '98: Proceedings of the fourteenth annual symposium on Computational geometry*. ACM, New York, NY, USA, 58–67.

FELKEL, P. AND OBDRZALEK, S. 1998. Straight skeleton implementation. In *Proceedings of Spring Conference on Computer Graphics*. 210–218.

GAL, R., SORKINE, O., MITRA, N. J., AND COHEN-OR, D. 2009. iWires: an analyze-and-edit approach to shape manipulation. *ACM Trans. Graph. 28*, 33:1–33:10.

HANLEY WOOD, LLC. 2010. eplans.com. http://www.eplans.com.

HAVEMANN, S. 2005. Generative mesh modeling. Ph.D. thesis, TU Braunschweig.

KELLY, T. W. A. 2006. City architecture generation. M.S. thesis, Bristol.

LAYCOCK, R. G. AND DAY, A. M. 2003. Automatically generating large urban environments based on the footprint data of buildings. In *SM '03: Procedings of the ACM symposium on Solid modeling and applications*. ACM Press, NY, USA, 346–351.

LEGAKIS, J., DORSEY, J., AND GORTLER, S. 2001. Feature-based cellular texturing for architectural models. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '01. ACM, New York, NY, USA, 309–316.

LIPP, M., WONKA, P., AND WIMMER, M. 2008. Interactive visual editing of grammars for procedural architecture. *ACM Trans. Graph. 27*, 102:1–102:10.

MARVIE, J.-E., PERRET, J., AND BOUATOUCH, K. 2005. The FL-system: a functional L-system for procedural geometric modeling. *The Visual Computer 21,* 5, 329–339.

MERRELL, P. AND MANOCHA, D. 2008. Continuous model synthesis. *ACM Trans. Graph. 27*, 158:1–158:7.

MICROSOFT CORP. Bing maps™. http://www.bing.com.

MÜLLER, P., WONKA, P., HAEGLER, S., ULMER, A., AND VAN GOOL, L. 2006. Procedural modeling of buildings. *ACM Trans. Graph. 25*, 614–623.

PRUSINKIEWICZ, P. AND LINDENMAYER, A. 1991. *The Algorithmic Beauty of Plants*. Springer Verlag.

STINY, G. 1975. *Pictorial and Formal Aspects of Shape and Shape Grammars*. Birkhauser Verlag, Basel.

WONKA, P., WIMMER, M., SILLION, F., AND RIBARSKY, W. 2003. Instant architecture. *ACM Trans. Graph. 22*, 669–677.

15. Manual d'usuari i/o instal·lació

En aquest capítol s'explicarà tot el que és necessari, tant d'instal·lació com de parametrització, per a que pugui funcionar el programa. Per altra banda s'explicarà com s'ha d'utilitzar a nivell d'usuari.

15.1. Instal·lació

Per a que pugui funcionar el programa del projecte a una màquina, es requereix primer de tot que es tingui instal·lat el Houdini, Python i Java, (a més del Python que té integrat el Houdini, es necessita instal·lar el Python al sistema operatiu).

Una altra cosa imprescindible és instal·lar el JPype, (fer-ho després de tenir instal·lat el Java, i l'assistent d'instal·lació ja localitzarà on està instal·lat el Java per introduir-hi la llibreria JPype)

Aleshores, per funcionar el programa cal ubicar els fitxers de manera que es puguin enllaçar mitjançant els paths que estan definits dins els codis font:

En total hi ha 2 fitxers independents: el ServerPRC.py i el Wrapper.class (el codi en Java ja compilat) i el codi que està al interior del node de Houdini, que està guardat dins la llibreria que s'ha creat amb el Houdini anomenada procExtr.otl. A més d'aquesta llibreria també es té la pròpia del buildingEngine, anomenada buildingEngine.otl. A part s'han de tenir totes les llibreries que necessita el mòdul de CampSkeleton. Per comoditat es podria ubicar tot dins una carpeta anomenada lib.[3]

Cal canviar dels codis per tal que puguin localitzar aquesta carpeta “lib”. En total s'han de fer 2 canvis.

- Dins el node de Houdini ProceduralExtrusions, caldrà localitzar la línia que es mostra a la Figura 15.1, que es troba a la funció engegarServer() dins la classe JPypeServer.

```
def engegarServer(self):
    import subprocess
    return subprocess.Popen('python C:/Marc/PFC/JPype/serverPRC.py')
```

Figura 15.1 Definició de Path dins el node de Houdini creat

A la Figura 15.1 es veu com s'està dirigint a la ruta: C:/Marc/PFC/JPype/serverPRC.py. Seguint el que s'ha dit anteriorment de tenir una carpeta lib on a dins es tinguin tots els fitxers, suposem que la carpeta està ubicada a la ruta C:\lib. La línia s'hauria de modificar de la manera com es mostra a la

3 Es tindrà una carpeta anomenada lib dins el disc dur que contindrà: els fitxers ServerPRC, Wrapper.class i totes les llibreries que necessita el mòdul de CampSkeleton.

Figura 15.2.

```
return subprocess.Popen('python C:/lib/serverPRC.py')
```

Figura 15.2 Com s'hauria de modificar la línia

- L'altre canvi s'ha de fer dins el fitxer ServerPRC.py. Cal localitzar la línia que es mostra a la Figura 15.3 dins la funció main().

```
def main(planta,perfils):
    jpype.startJVM(jpype.getDefaultJVMPath(),
    "-Djava.class.path=C:/Marc/workspace/PFC/bin/","-Djava.ext.dirs=C:/Marc/PFC/lib/")

    pkg = jpype.JPackage('com')  # get the package
```

Figura 15.3 Definició de Path dins el fitxer ServerPRC

En aquesta comanda del JPype hi ha 2 atributs on es defineixen els paths. El primer és per localitzar el fitxer Wrapper.class, i l'altre per localitzar totes les llibreries que utilitza el mòdul de CampSkeleton. Com que en aquest exemple es tindran dins una carpeta anomenada lib, als 2 paths s'introduirà el mateix. Veure Figura 15.4.

```
jpype.startJVM(jpype.getDefaultJVMPath(),
"-Djava.class.path=C:/lib/","-Djava.ext.dirs=C:/lib/")
```

Figura 15.4 Com s'hauria de modificar la línia

15.2. Manual d'usuari

Seguint el diagrama d'activitats que s'ha presentat al capítol 8 d'anàlisi i disseny del sistema, primer de tot cal inicial el Houdini. Un cop obert s'ha d'instal·lar la llibreria Procedural Extrusions que està integrada al Houdini, se suposarà que està a la carpeta lib ubicada en C:\. Veure Figura 15.5.

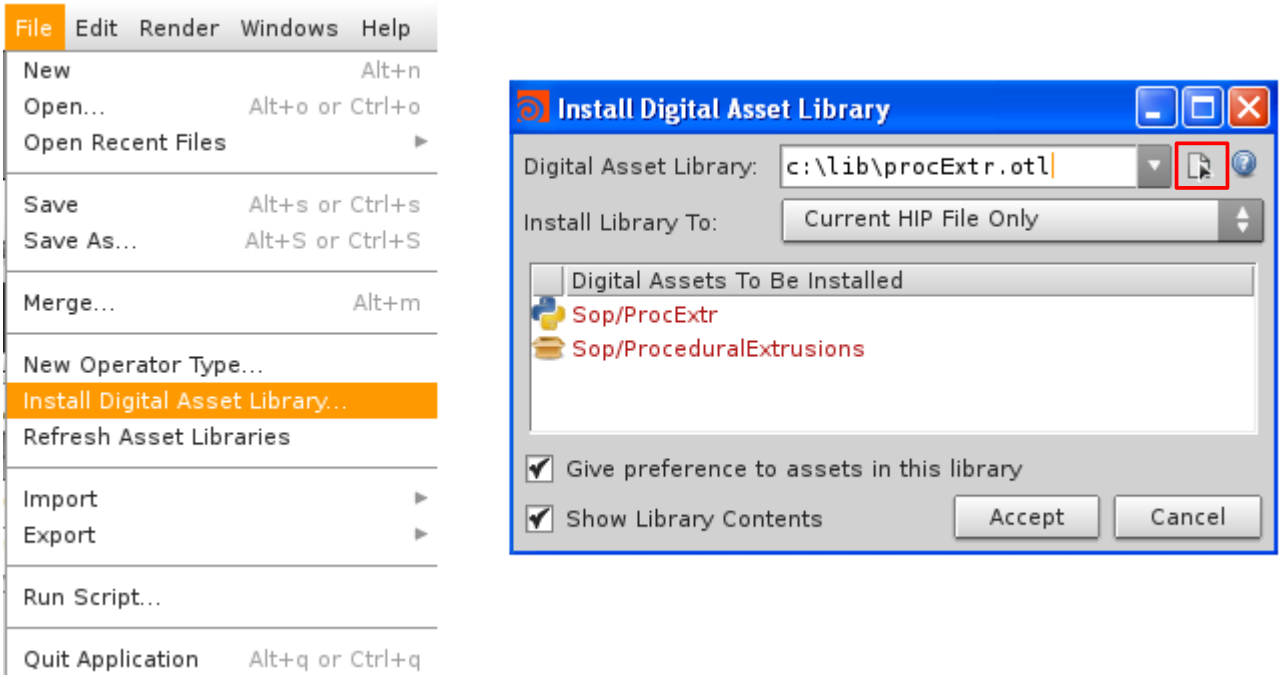


Figura 15.5 Com instal·lar la llibreria Procedural Extrusions dins el Houdini

Tal i com es mostra a la Figura 15.5, s'ha d'anar a “File”, “Install Digital Asset Library...”, i s'obrirà la finestra de la dreta de la Figura. Dins aquesta Figura es pot introduir directament la ruta on es troba la llibreria o sinó apretant l'icona de la dreta del quadre on s'introdueix la ruta. En aquest cas, s'obrirà una altra finestra on es podrà explorar tot el disc dur. El nom de la llibreria és procExtr.otl.[4]

A part d'instal·lar el Procedural Extrusions, també cal instal·lar el buildingEngine per si després d'haver creat l'edifici es vol introduir algun accessori. Es fa de la mateixa manera però localitzant el fitxer buildingEngine.otl.

Un cop realitzada tota aquesta tasca, ja es pot començar a dibuixar amb el Houdini. Primer de tot s'ha de crear un node Geometry per tal que a dins d'aquest es pugui crear la geometria de l'edifici.

El següent pas serà definir una planta i un o varis perfils, per comoditat es recomanaria començar per dissenyar la planta. Veure Figura 15.6.

4 Només cal instal·lar les llibreries quan es comença un projecte de nou. Si es treballa dins un projecte on ja s'han instal·lat, i es guarda, un cop es torni a obrir, les llibreries ja estaran instal·lades.

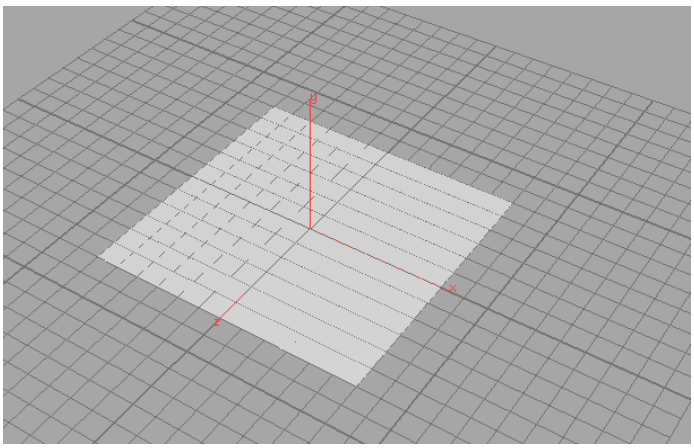


Figura 15.6 Exemple de disseny de planta

Per dissenyar una planta, tal i com es mostra a la Figura 15.6, es fa mitjançant el node Curve. Recordar que al definir una planta, només pot estar continguda en pla XZ, (horitzontal al terra).

De la mateixa manera es defineixen els perfils, amb el node Curve. Aquests han d'estar definits sobre el pla XY, de la següent manera, veure Figura 15.7.

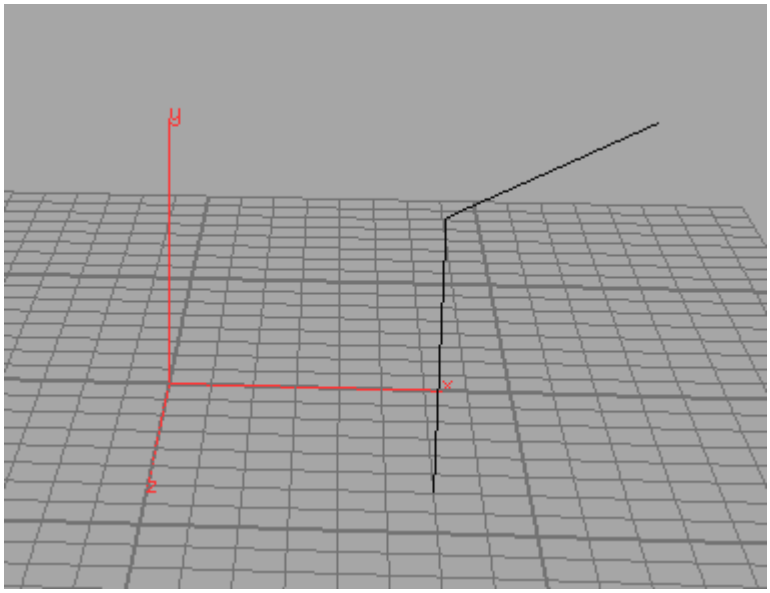


Figura 15.7 Exemple de disseny de perfil

En el cas del perfil mostrat en la Figura 15.7, la façana tindria una forma totalment vertical fins a un punt on s'inclinaria amb direcció al cap al centre de l'edifici amb la inclinació que s'hagi definit (donada pel vèrtex on acaba de pujar verticalment i el vèrtex següent).

Un cop definida una planta i un perfil, els nodes es troben en la situació descrita a la Figura 15.8.

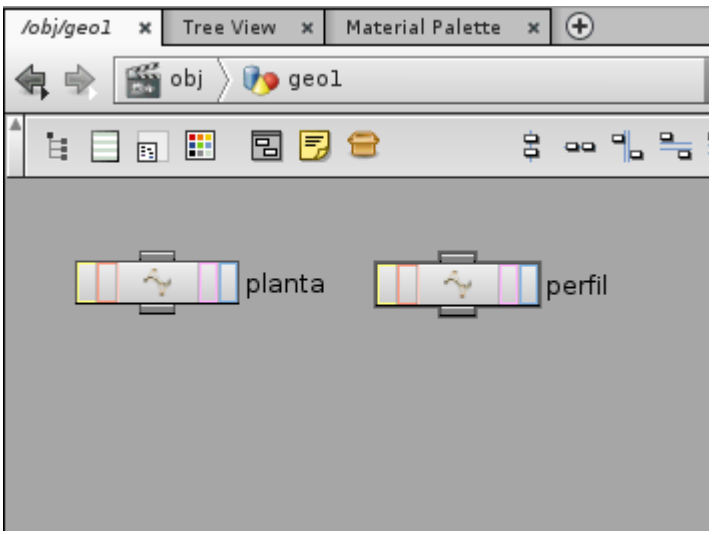


Figura 15.8 Un cop definides planta i perfil

El següent pas seria crear el node Procedural Extrusions i enllaçar-hi la planta i el perfil. És important tenir clar que, l'entrada de l'esquerra és exclusiva per la planta, i la de la dreta pels perfils. Veure Figura 15.9.

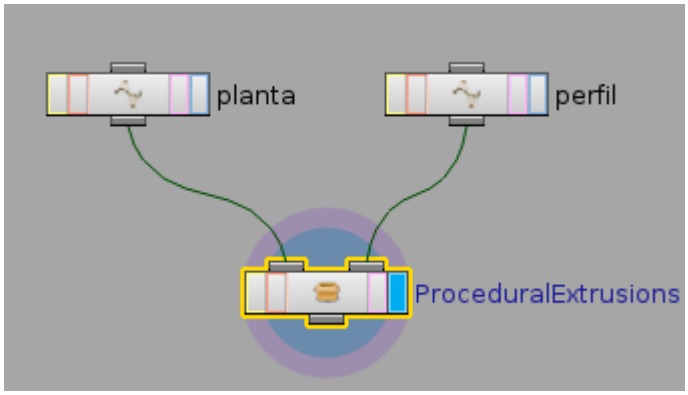


Figura 15.9 Estructura dels nodes per crear l'edifici

15.2.1. Definir més d'un perfil

Fins aquí s'ha explicat com crear un edifici a partir d'una planta i un perfil, ara s'explicarà com poder definir més d'un perfil i com associar-los a cada façana que tingui l'edifici.

Cada façana de l'edifici busca per tot l'espai geomètric, quin perfil està situat més aprop del punt mig d'aquesta, de manera que fa un recorregut per tots els perfils que hi hagi definits i es guarda el que estigui més aprop, aquest serà el perfil que utilitzarà aquesta

façana, i això passa amb cada façana.

Si es vol definir més d'un perfil per a l'edifici, existeixen 2 maneres de treballar: la més simple és anar creant tants nodes “curve” com perfils es vulguin definir. L'altre manera és definir un sol node “curve” i seguit d'aquest se li enllacen nodes “transform” de manera que la sortida de cada node “transform” és un nou perfil.

Per associar els perfils a les arestes de la planta es farà mitjançant distàncies, o sigui per cada aresta de la planta es buscarà el perfil que estigui més a prop de tots els que l'usuari hagi definit.

A les Figures 15.10 i 15.11 es pot veure com quedaria un edifici definint 2 perfils diferents.

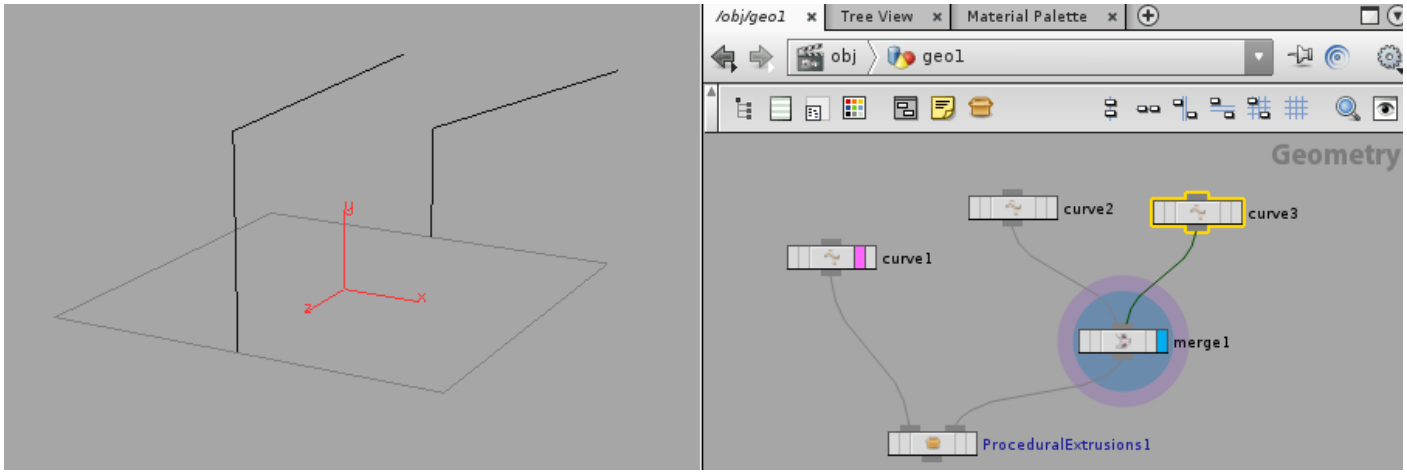


Figura 15.10 Definició de més d'un perfil

Tal i com es veu en la Figura 15.10, per definir 2 perfils, s'han creat 2 nodes curve, i llavors s'han unit amb el “merge” per connectar-ho amb el node de ProceduralExtrusions.

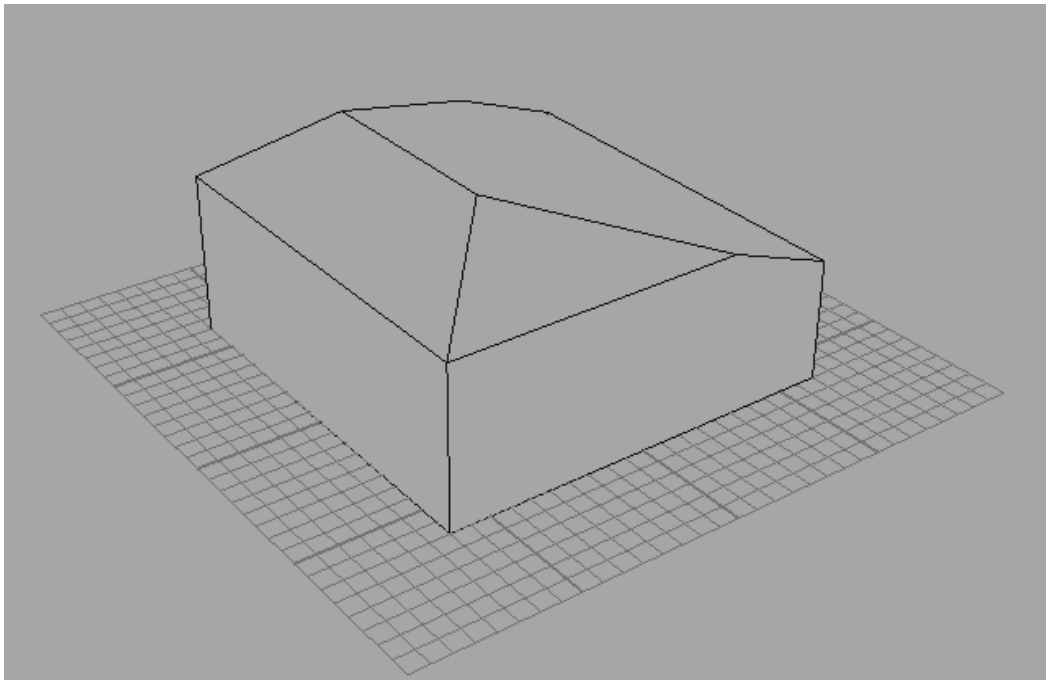


Figura 15.11 Exemple resultat edifici amb 2 perfils