Masters Program in **Geospatial Technologies**

3d GIS modeling using ESRI´s CityEngine

A case study from the University Jaume I in Castellon de la Plana Spain

Kristinn Nikulás Edvardsson

Dissertation submitted in partial fulfilment of the requirements for the Degree of *Master of Science in Geospatial Technologies*

UNIVERSIDADE NOVA ISEGI
UNIVERSITAT JAUME·I
WESTFÄLISCHE WILHELMS-UNIVERSITÄT MÜNSTER

3d GIS modeling using ESRI´s CityEngine

A case study from the University Jaume I in Castellón de la plana Spain

Dissertation supervised by

PhD Michael Gould

And co supervised by

PhD Joaquin Huerta

PhD Marco Painho

March 2013

Acknowledgments

I wish to thank my supervisors Michael Gould, Joaquin Huerta and Marco Painho for making this exciting project a reality and for the opertunity to work with CityEngine. I would like to thank my fellow masters students for the time spent together working on our projects and for experiences that I will never forget. Last but not least I want to express my gratitude for Andres Munos and all the staff working at the University Jaume I for their support and patience.

3d GIS modeling using ESRI´s CityEngine

A case study from the University Jaume I in Castellón de la plana Spain

ABSTRACT

The field of GIS has in recent years been increasingly demanding better 3d functionality and ever more resources is being spent in researching the concept. The work presented here is on the integration and use of procedural modeling language for generating a realistic 3d model of the University Jaume I in Castellón de la Plana Spain. The language used is a modified version of shape grammars combined with split grammars with the software CityEngine. The work is conducted with the use of CityEngine and is intended as a contribution to the currently active research on the use of shape grammar in GI science.

Smíði LUK þrívíddar líkans

Ferilsathugun framkvæmd á háskólasvæði University Jaume I í Castellon de la plana á Spáni

Útdráttur á Íslensku

Þróun inann landfræðileg upplýsingakerfa hefur á allra síðustu misserum stöðugt kallað eftir bættum aðferðum til að búa til þrívíddar model. Hér verður lagt mat á þá þróun sem hefur verið í gangi undanfarið við þróun hugbúnaðar sem nýtir "procedural" forritunarmál til að smíða þrívíddarmódel. Rannsóknarsvæðið er háskólasvæði University Jaume I í Castellón de la Plana á Spáni. Forritunarmálið er byggt samanstendur af "shape grammars" og "split grammars" og grunnurinn í foritinu CityEngine sem notast verður við. Lagt verður mat á notkun þessa hugtaka við smíði stafrænna þrívíddarmódela frá sjónarhóli landfræðinga.

KEYWORDS

GIS Applications

City Models

3d

CityEngine

Shape Grammars

Split Grammars

Procedural modeling

Urban Planning

ACRONYMS

CAD – Gomputer-aided Design

CGA – Computer Generated Architecture

GIS – Geographical Information Systems / Sience

MTL – Material Template Library

MTL – Material Template Library

OBJ – Object File

UJI – Universidad Jaume I

ViscaUJI – Virtual Campus for the University Jaume I

INDEX OF THE TEXT

Table of Contents

INDEX OF TABLES

INDEX OF FIGURES

1. INTRODUCTION

The field of GIS is currently undergoing a major shift in visualization technology. With the oncoming of ever more powerful computers the restrictions of the 2d mapping traditions no longer apply. Although the history of GIS and computer mapping systems date back to the 1960s (Coppock et. al., 1991), it is not until recently that digital maps have been elevated to 3d (Elwannas, 2011).

The first devices to commercially utilize 3d maps have been navigation devices such as handheld GPS systems, navigators for cars and airplanes, although not full 3d but usually defined as 2.5d.

The main driving force behind 3d development has been the computer gaming and animation industry and now we are witnessing the merging of those kinds of computer graphics with GIS. One of the biggest GIS software developer ESRI is currently taking a big step towards full 3d visualization by acquiring one of the leading companies in development of 3d urban modeling software Procedural. In 2008 Procedural released the software CityEngine for the creation of 3d cities in a quick and relatively simple manner using procedural modeling language. The program uses procedural modeling methods combined with shape and split grammars for generation of 3d content.

The goal ESRI has set itself is to make CityEngine a part of their ArcGIS family of applications. Jack Dangermond president of ESRI stated at the ESRI's 2011 International User Conference: *"Many GIS problems can only be solved in 3D, particularly in the area of urban development, Procedural's unique capabilities for generating high-quality 3D data, using the same GIS data our users already have, makes them a perfect match for Esri."* (Handrahan, 2011).

The main focus of this research paper is on testing the latest version of CityEngine published by ESRI in 2012 for 3d map making purely from a GIS perspective. The goal is firstly to look into CityEngine and see if it can in fact be used to generate 3d models of a "real city". The dataset is a conventional 2d GIS data stored in an ESRI file geodatabase derived from a CAD architectural drawing of the area. It has been imported into a file geodatabase that is based on the local government template, for further information see (ArcGIS Resouces, 2013b).

Modeling a whole city however is beyond the scope of this thesis; the study area presented here is the university Campus area of University Jaume I in Castellón de la Plana Spain. A

brief experiment will also be conducted with indoor mapping using CityEngine to see if the software can be used for that purpose.

Figure 1: An overview map of the UJI Campus.

In the course of this research multiple issues came up in relation to modeling existing GIS datasets with CityEngine. Hopefully the experience of the authors as geographers and GIS users of many years will be beneficial to further development of CityEngine.
The project has been divided into two parts and this part will focus on following issues:

- Modeling of the campus basemap

  The basemap will be made up of three datasets, street network (lines), landscape areas (polygons) and Streets and pavements (polygon).
- Indoor mapping

  The indoor map was created for the Rectorate building on the campus area from interior spaces polygon layer and building floorplan line layer.

2. Theoretical Framework

It has been stated that spatial datasets are in some way special, that they display certain characteristics that are different from other types of data. The most obvious example of this is Toblers first law of geography: "*everything is related to everything else, but near things are more related than distant* things" (Tobler, 1970). The specialty of geographical data has prompted a whole new software industry focusing on specifically on it, the GIS industry. The industry traces its roots back to the 1960s when it the Atlas of Great Britain and Northern Ireland revealed that the only way to efficiently manage geographical data is with the help of computer databases. (Coppock et. al., 1991).

In the 1980 the software producer ESRI got the lead on designing GIS software by exploiting the relational database model for storing geographical data as vector data. This model worked well for environmental and resource management but as the technology advanced there was an increased demand for new data types that did not fit well into the current model. Urban planners for example were looking for ways of utilizing GIS within their profession (Zlatanova, et. al., 2002). At the same time new database technology was influencing database design, the shift was from relational databases to object orientated databases. Topology is one of the things that make GIS data special and is directly related to Toblers first law of geography. The new object orientated database model had a higher computational cost so topology was set up in such a way that the user has to choose if he wants to use it or not. The older systems could do this on the fly but the benefit of this design was particularly important when it comes to 3d since the typical 3d computer model is composed of objects. (Goodchild, 2003).

The new database systems designs attempted to meet the new demands of the market but these experiments newer fully materialized into a full 3d system and thus the term 2.5d was coined for 3d. GIS. Systems such as ArcGIS Spatial Analyst are a good example of 2.5d GIS software and with it users can for example generate surfaces, compute volumes drape raster images and evaluate visibility from point to point to name a few. These functionalities have been hampered by the fact that these objects can only handle one z value (Stoter et. al., 2003).

6.1. Looking ahead

Today there is an ongoing race among designers of GIS software's for coming up with the best functioning full 3d GIS software. To achieve this two main issues must be overcome:

1. How is 3d topology implemented?
2. What type of data structure is best suited for 3d objects?

Sisi Zlatanova points out that *"The most critical difference of GIS compared to other software has always been the possibility to perform spatial analysis and visualize them. This means practically that the models (topology, geometry, network, spatial occupancy enumerations, free form surfaces etc.) have to be first agreed upon"* (Zlatanova, 2009). This has not happened yet although we have seen some development taking place among separate developers.

2.1.1. Data structures and types

So far there has not been a universally adopted data type for 3d GIS in the same way as the ESRI shape file vector model became the unofficial standard for 2d GIS data. There have been numerous researches on the issue and at least 14 different types of data have been proposed. They can be organized into four main categories:

- Representation 3d objects by its boundaries.
- Representation 3d objects by voxel elements.
- Representation 3d objects by a combination of the 3d basic block.
- Combined models.

The representation of 3d objects by its boundaries model is the data type that has been most successful in 3d GIS. It has been adopted in projects such as CityGML and used in the ArcGIS multipatch data type. The Principle is that objects are based on elements already defined, such objects can be: Point, Line, Surface, and Body. Lines can be straight, arcs or circles and surface may be flat. Volume is an extension of the surface for representing 3D blocks; the blocks can be: box, cone, cylinder or combination of all three. (Tuan, 2013). These objects are then lined up together to form a 3d object and they can then be assigned colors or textures, this method has been the basis of 3d graphics for a long time. The data type is the same one as CAD datasets are based on but instead of using local coordinates they

utilize geographical coordinates (Nilsen, 2007), when used solely for the purpose of visualization this method is very effective. When on the other hand we need to design multiple objects that all have to have a single unique identifier for querying and other analysis the situation becomes more complicated and has not yet been sufficiently solved.

2.1.1. The multipatch

Today most 3d GIS models lack functionalities such as querying 3D objects, and 3D analyses such as overlay, 3D buffering and 3D shortest route (Stoter et. al., 2003). This is due to the way 3d objects are generated with the polygonal modeling technique. The standard way has been to build up 3d models from geometrical objects and link them together in a local coordinate system; as a result a single cube for example consists of 12 polygons. It then becomes obvious that to setup a query on a house that is made up of 12 or more polygons becomes complicated.

This issue becomes apparent when one thinks about topological relation of such an object, one polygon of the house might be located inside another but not the remaining. Makers of GIS software have been faced with this problem for some time now and GIS vendor ESRI has come up with a promising solution to this issue, the multipatch. The multipatch is a traditional 3d object made up of triangles, circles and arcs, the shapes are connected inside the geodatabase and as a result the 3d object appears as a single object in the attribute table. In theory this is a solid 3d object but unfortunately topology has not been developed for this model yet in the current 10.1 release of ArcGIS (ArcGIS Resouces, 2009).

Once you have a 3d solid object in the database it should be relatively easy to start building tools for analysis, this has not happened either unless to a very limited extent. 3d analysis tools have until now been centered on visualization rather than numerical data with tools such as skyline analysis and others.

2.1. Mass 3d modeling

Another issue connected to 3d GIS is the time issue when creating a 3d model. It is obvious that the method of modeling a building in CAD based tool such as Trimble SketchUp (formerly Google SketchUp) and uploading them to the web is not an economical way generating a large city models for analysis (Muller et. al., 2006). ESRI has increasingly been

looking into solving this problem by looking at the 3d gaming and animation industry. The industry has for a long time been demanding a tool that can create a realistic 3d model of a city on a large scale efficiently.

In paper from 2001 Pascal & Yoav presented a new way of modeling whole cities in a semi automatic way with software they called CityEngine. CityEngine uses modified L-systems to grow street networks; the space in between them is subdivided into lots which are then further subdivided into shapes. Such a network can be set up in a few minutes with the automation process but if the user wants to influence it he can create the network manually. The shapes and the street network are then assigned specific set of shape grammar rules that define the shape and texture of the buildings that will be generated from the shapes (Parish & Müller, 2001). The use of shape grammars is ideal for the generation of 3d GIS models since they can take 2d shapes and generate a 3d shapes from them. CityEngine natively supports the import of the ESRI shape files and other geographical datasets such as rasters, this ensures that existing 2d data will not have to be redesigned for the 3d use.

2.2. Shape grammar

The first formal publication on shape grammars dates back to 1972 in an article by Stiny, G. and Gips, J. Shape grammars perform computations with shapes in two steps: recognition of a particular shape and its possible replacement. Rules specify the particular shapes to be replaced and the manner in which they are replaced. Underlying the rules are transformations that permit one shape to be part of another. The original concept was used for painting and sculpting objects, it proved to be a success and since then the concept has been in continuous development (Stiny et. al., 1972). Pascal Muller and Yoav I. H. Parish experimented with the idea of applying shape grammars to 2d polygons for extrusion into 3d space and assigning them textures. This approach demonstrated how a whole city could be modeled in few minutes without the need of modeling each building separately (Parish et. al., 2001). A real breakthrough occurred in 2003 when Wonka and others introduced the concept of split grammars. In its simplest form split grammars split a 3d object into its components such as faces, edges or vertices. By combining the split grammars with shape grammars they developed a new method in modeling buildings (Wonka et. al., 2003). The culmination of this work resulted in the software program CityEngine commercially released in 2008. The

program took advantage of the split and shape grammars for procedural modeling of 3d architecture from 2d polygons.

3. CityEngine

CityEngine has established itself well within the gaming industry since its commercial release in 2008. The software has been used extensively to create highly detailed 3d model of fictional cities and urban landscapes. CityEngine has the option of importing various forms of geographical and architectural datasets such as GIS and CAD, its possibilities of modeling real landscape are quite promising. However the most efficient and widely utilized way to use the software has been to start an empty scene and let the software grow the landscape with minimal user input. After growing the streets CityEngine creates lots in between them and then divides the lots into polygon shapes (figure 1). The user can then assign a CGA (computer generated architecture) rule file to each of these polygons shapes. The CGA rule file is the actual definition of each building where it is defined with shape and split grammars.

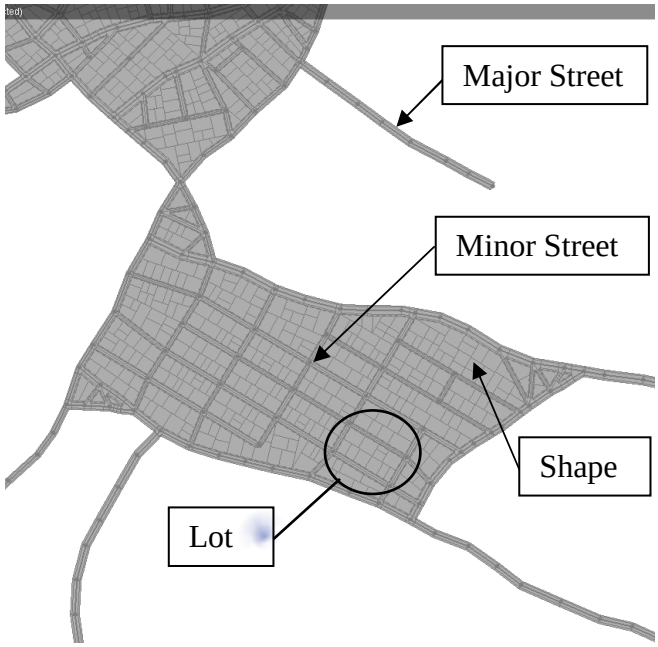

Figure 2: Streets grown in CityEngine.

There are three types of patterns the user can select when generating a city this way: Organic, raster and Radial. The Organic method creates streets that have the characteristic of old medieval city and look like small clusters that grew into a bigger city. The Radial method is based on cities like Paris where the city evolved from a centre point, not uncommon where cities evolved as a fortress surrounded by a city wall. The last one the raster option create a city that looks like it was planed from the very beginning, where parallel streets cross at more or less 90° angle, New York has been used as an example of such a city. The user also chooses from these patterns a pattern for minor streets that surround the lots where the building will be created. The lots can then have recursive subdivision, skeletal subdivision, offset subdivision or no subdivision depending on the style the user wants to achieve. The user can also use height maps and obstacle maps to limit the city; the use of population data is also acceptable to influence street growth patterns (Maren et. al., 2012; Parish. et. al., 2001).

2.1. The user interface

CityEngines user interface (figure 3) is a well laid out and designed for use on a large or multiple screens. The software has many advanced functionalities running under the surface while the user interface is relatively user friendly. This chapter provides a short description of the user interface and the most important components used to generate the model. Figure 3 shows a screen shot of the main user interface and how its components are laid out and will be used in the following short description.

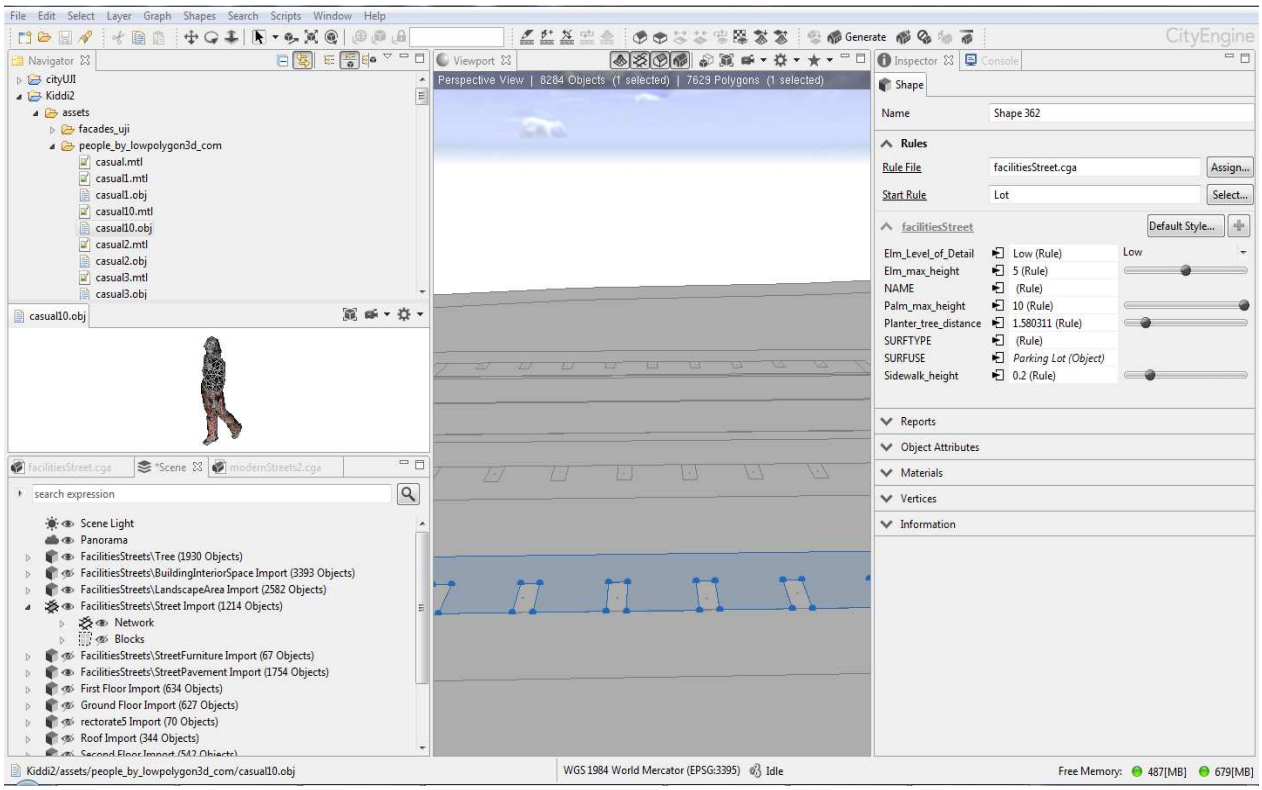

Figure 3: The default user interface of CityEngine.

3.1.1. The navigator

In the top left corner of the user interface is the navigator which is CityEngines file manager. The root of the file manager is the workspace folder and when a new CityEngine project is created a folder is created in the workspace with standard CityEngines folder structure.

3.1.2. The preview window

Below the navigator is the preview window where the user can preview all data formats that city engine can read. Examples would be 3d models like .obj files, most image formats including .jpeg, .tiff, .png and various other such as shapefiles and ESRI file geodatabases.

3.1.3. The scene editor

Below the preview window is a tabbed space that holds the scene editor among other interfaces. When the user opens a scene the different layers of the scene are displayed in the scene editor tab. The user can expand each layer to display a list of all its shapes and networks, also all static models are accessible here.

3.1.4. The rule editor

The rule editor is displayed as a different tab in the same tab space as the scene editor. It is a simple text editor for creating and modifying CGA rules but can also be used to edit all text files such as .txt or .mtl.

3.1.5. The viewport

One of the most important part of the user interface is the viewport located at the middle of the screen. This is the window where the user can visualize and manually edit the models. In the viewport shapes can be selected for editing and CGA rule assigning. New shapes are created here and imported shapes can be transformed to static model for manual editing. The viewport has several different viewing perspectives and users can also define visual effects but the more graphics the slower the performance.

3.1.6. The inspector tab group

At the far right of the user interface is another tab group that contains the inspector and other interfaces. The inspector allows the user to collect information on the selected shapes such as vertices, materials and more, but most importantly the object attributes and all rule attributes assigned to it. The inspector is also used to assign the CGA rules to shapes.

3.1.7. The console and scripting

In the far right tab space is also the console window and most usable part of it the python console. The python console is a useful tool to run scripts most notably if the user wants to select objects by attributes. There is no tool for selecting shapes by attributes but this can be achieved with the python console by running the following script:

```
ce = CE()

def selectByAttribute(attr, value):
    objects = ce.getObjectsFrom(ce.scene)
    selection = []
    for o in objects:
        attrvalue = ce.getAttribute(o, attr)
        if attrvalue == value:
            selection.append(o)
            ce.setSelection(selection)

if __name__ == '__main__':
    pass
```

To simplify the process of running these scripts a helper script file can be created with the text editor and it should be saved at default script location (workspace/scripts), it can then be linked to from the python console. The above code should be saved as .py file in the script folder then the path to it is loaded into the system as follows:

```
sys.path.append(ce.toFSPath("scripts"))
```

The script is then imported:

```
import scriptName
```

Now the system is setup for selection by attribute using the code:

```
scriptName.selectByAttribute("attributeName",
"attributeValue")
```

The python console can be used to perform various other functions such as generating models and more but those functions will not be used in this project.

4. Methods

This chapter is divided into three main sections; the first one is on the projection of the ViscaUJI geodatabase and how to make it compatible with CityEngine. The second one is on the creation of a base layer for the campus grounds and the third chapter is on the creation of an indoor map of the Rectorate building at the UJI campus.

4.1. Setting up the projection

The ViscaUJI geodatabase coordinate system is the WGS 1984 Web Mercator Auxiliary Sphere with the EPSG code 3857 which is currently not supported by CityEngine. The solution is to convert the data to a coordinate system that CityEngine supports. The closest one is the WGS 1984 World Mercator coordinate system with the EPSG code 3395. The simplest way to do this is create a new file geodatabase that contains all the feature datasets we plan to use in our 3d model using in ArcCatalog. For the fastest conversion new database is created containing a feature dataset named FacilitiesStreets with the WGS 1984 World Mercator coordinate system. The following datasets are then imported from the ViscaUJI into it:

1. FacilitiesStreets/LandscapeArea.shp (Original).
2. ViscaUJI/FacilitiesStreets/Tree.shp (original).
3. ViscaUJI/FacilitiesStreets/StreetPavement.shp (needs to be modified first, see chapter 4.2).
4. ViscaUJI/FacilitiesStreets/BuildingInteriorSpace.shp (original).
5. ViscaUJI/FacilitiesStreets/BuildingFloorplanPublish.shp (needs to be modified first, see chapter 4.3.1).

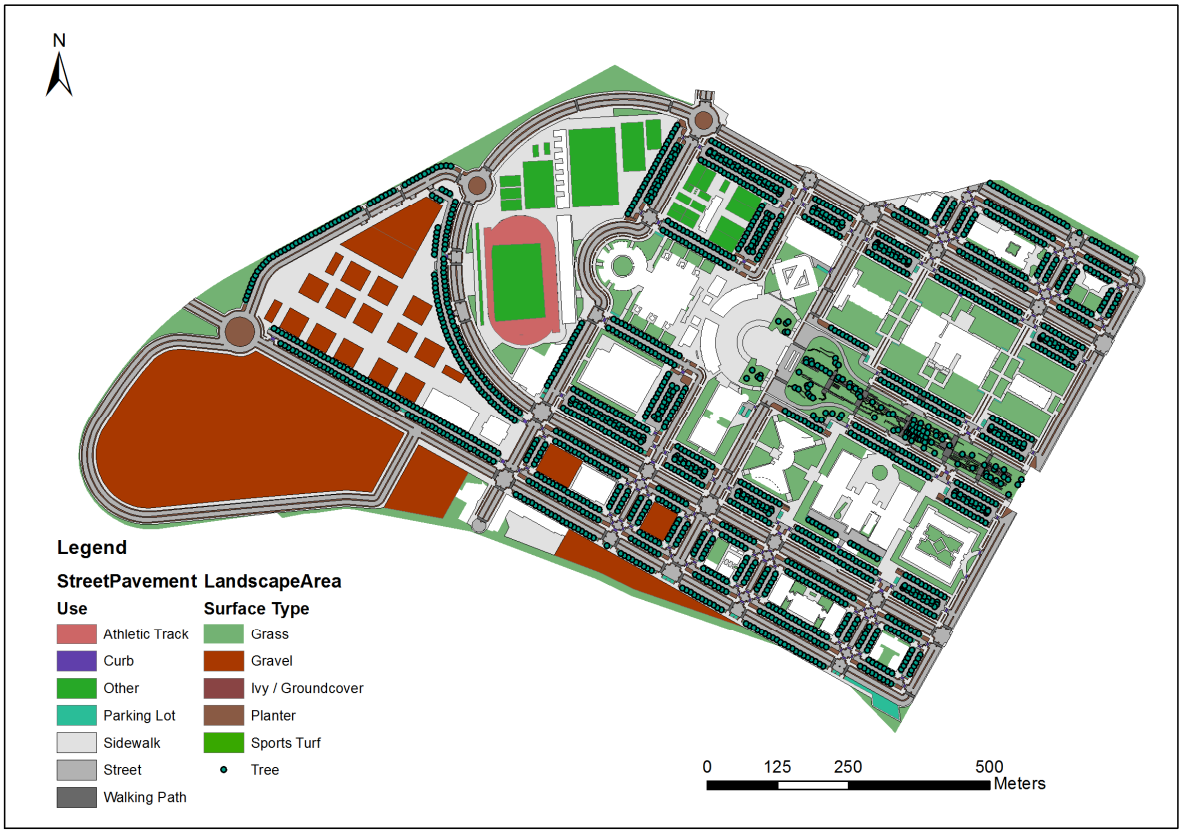

Figure 4: The datasets used in the base layer.

4.2. The base layer

ViscaUJI has two main polygon layers that make up the base layer for the campus grounds. They are the LandscapeArea and StreetPavement contained within the FacilitiesStreets dataset. They can be imported directly into CityEngine using the GDB importer, but before doing so some modifications must be made to it. The dataset has to be added to ArcMap and the following adjustments have to be performed on it:

The layer has a polygon type called streets and there are 12 polygons of the street type that have holes in them (see figure 6). CityEngine automatically fills up these holes when the polygons are imported and they need to be cut in half before they can be imported.

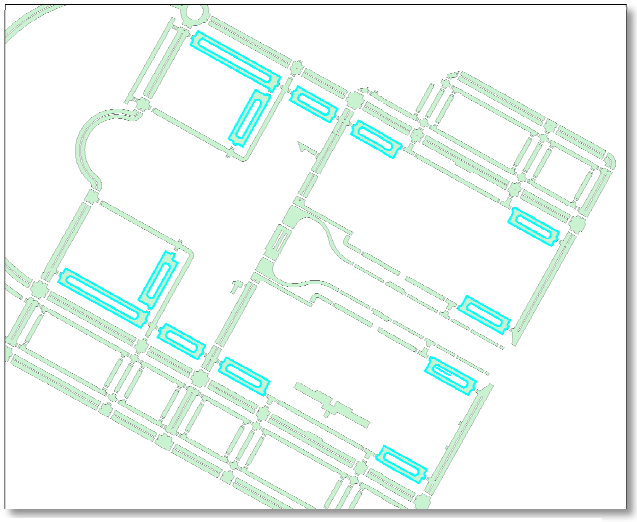

Figure 6: The 12 polygons that need to be cut

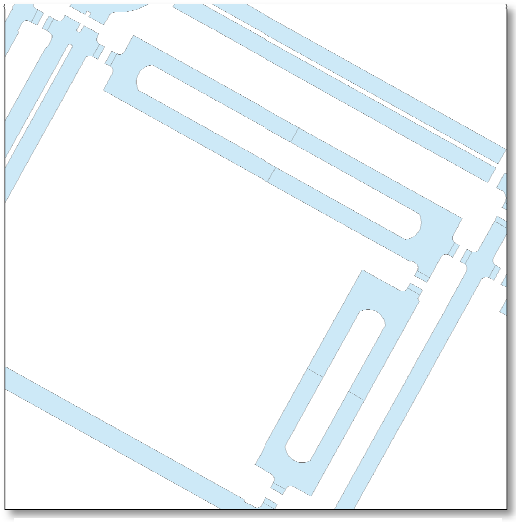

Figure 5: The polygons after the cut operation

The street dataset in now ready for CityEngine and the next step is to start it up and go to file, select new CityEngine Project and click next. The project should be assigned a name which in this case is SmartUJI and click finish.

There are several ways of importing a geodatabase into CityEngine, the method that works best is by copying the database to the data folder of the project. Then go to the navigator window open up the data folder (it might need refreshing after copying the data) right click the database and select import. CityEngine does not automatically detect the data type so the file GDB import must be chosen and then click next. Select all the data and from chapter 4.1. and click next. The next screen presents several cleanup options and none of them should be selected, if selected CityEngine will try to merge vertices to make them less complex but this will distort the model.

In this project a mixed method for the basemap model is used by combining polygon layers with a street network layer. This method was chosen because it is very difficult to apply textures to the street polygons of the StreetPavement layer in order to maintain correct centerline and other markings on polygon layers. City Engine can deal with streets much more efficiently if they are represented as networks using the modern streets template.

After analyzing the data visually it becomes clear that there are indeed three types of streets on the UJI campus all categorized as one data type named street:

- Major roads, made up of four lanes two in each direction and in some cases lead into roundabouts.
- Minor roads, made up of two lanes one in each direction.
- Parking space roads, no lanes.

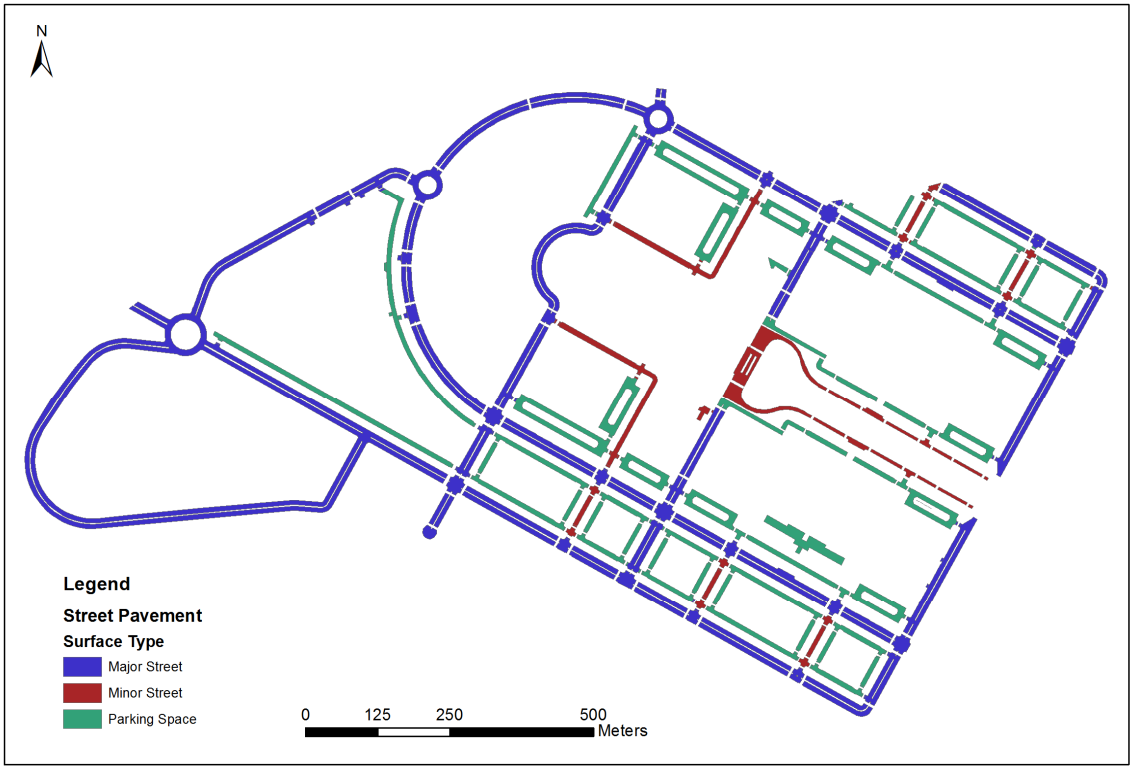

Figure 7: The street classification applied to the UJI Campus

The first two types in figure 7 are exactly the once the modern streets dataset can model but the roads in the parking spaces have no chance of being modeled by it. This is due to fact that in some cases they have angles and shapes that cannot be interpreted by a line. On the other hand the roads in the parking spaces have no centerline or other markings so they can be assigned an asphalt textures later on. The decision was then made to delete all polygons representing major and minor roads from the StreetPavement layer and create a graph network in the empty spaces. In the LandscapeArea layer there so called planters in the center of the major roads and they also need to be deleted since the modern streets template generates these planters as parts of the street network graph.

4.2.1. Building the street network graph

Now a new network graph has to be created from the layer menu of CityEngine and it should be digitized in the middle of the empty spaces where the street polygons used to be. The width of each segment has to be set to a size that overlaps or touches the sides of the planters in the LandscapeArea layer. It is critical that the width overlaps the edges just barely and the sidewalks left and right attributes be set to 0 since they will be generated later on. Table 1 gives an overview of a typical street segment attribute for network graph, figure 9 is an example of how the resulting graph segment should look like.

Table 1: Typical attributes for the graph networks segments

Attribute	Major road	Minor road
`shapeCreation`	`true`	`true`
`streetWidth`	`17.8`	`9.1`
`sidewalkWidthLeft`	`0`	`0`
`sidewalkWidthRight`	`0`	`0`
`precision`	`0.5`	`0.5`

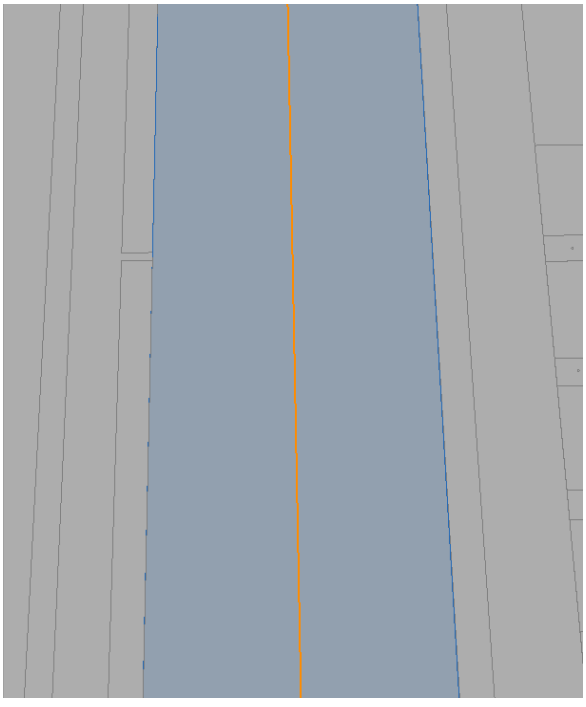

Figure 9: Example of how the street network graph must fit tightly inside the LandscapeArea layer.

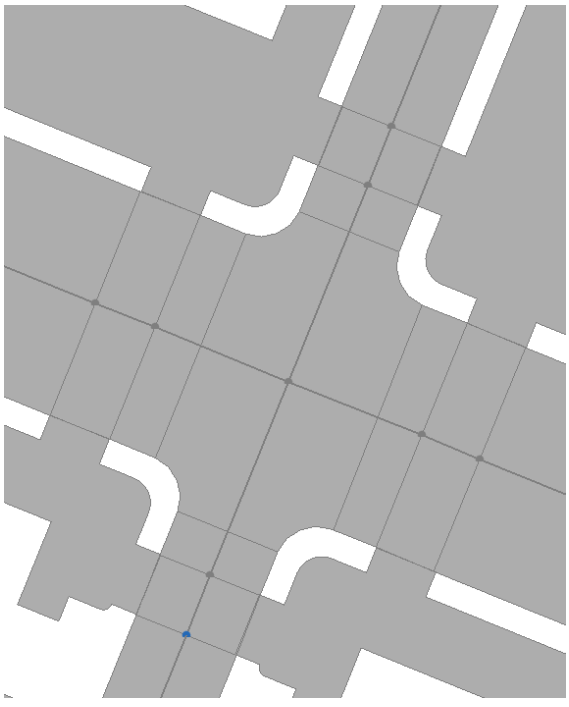

Figure 8: demostrating how vertices must be added where the pavement touches the streets.

It is desirable that the vertices of the network graph are as few as possible although multiple vertices are unavoidable everywhere the streets have curves since city engine does not deal effectively with curves in graph networks. In all locations where the pavement polygon touches the street is a walking path and at those locations vertices must frame the path as seen in figure 8.

The `minArcRadius` attribute of all intersection nodes must be set to 4 so that edge curves fit well to the Street pavement layer. Roundabouts are interpreted by a single network node instead of a full digitized street; there are three roundabouts on the campus ground that this applies to. The attributes of these three nodes must be set to the following values to resemble the real situation as accurately as possible:

```
shapeCreation   :    true
type            :    roundabout
precicion       :    0.75
minArcRadius    :    4
cornerStyle     :    Arcs (default)
innerRadus      :    15.5
streetWidth     :    10.5
```

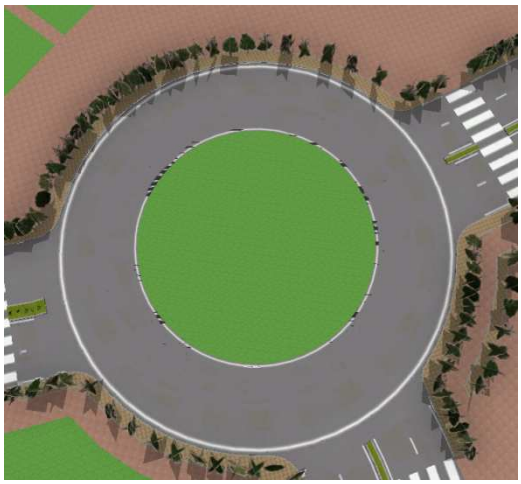

Figure 10: CityEngine generated roundabout.

The completed network graph is shown in Figure 11, on inspection notice the small dead end street that connect to the parking lot. They should have the same street width as the minor roads and just barely overlap the parking space road polygon as it will be layered on top of it.

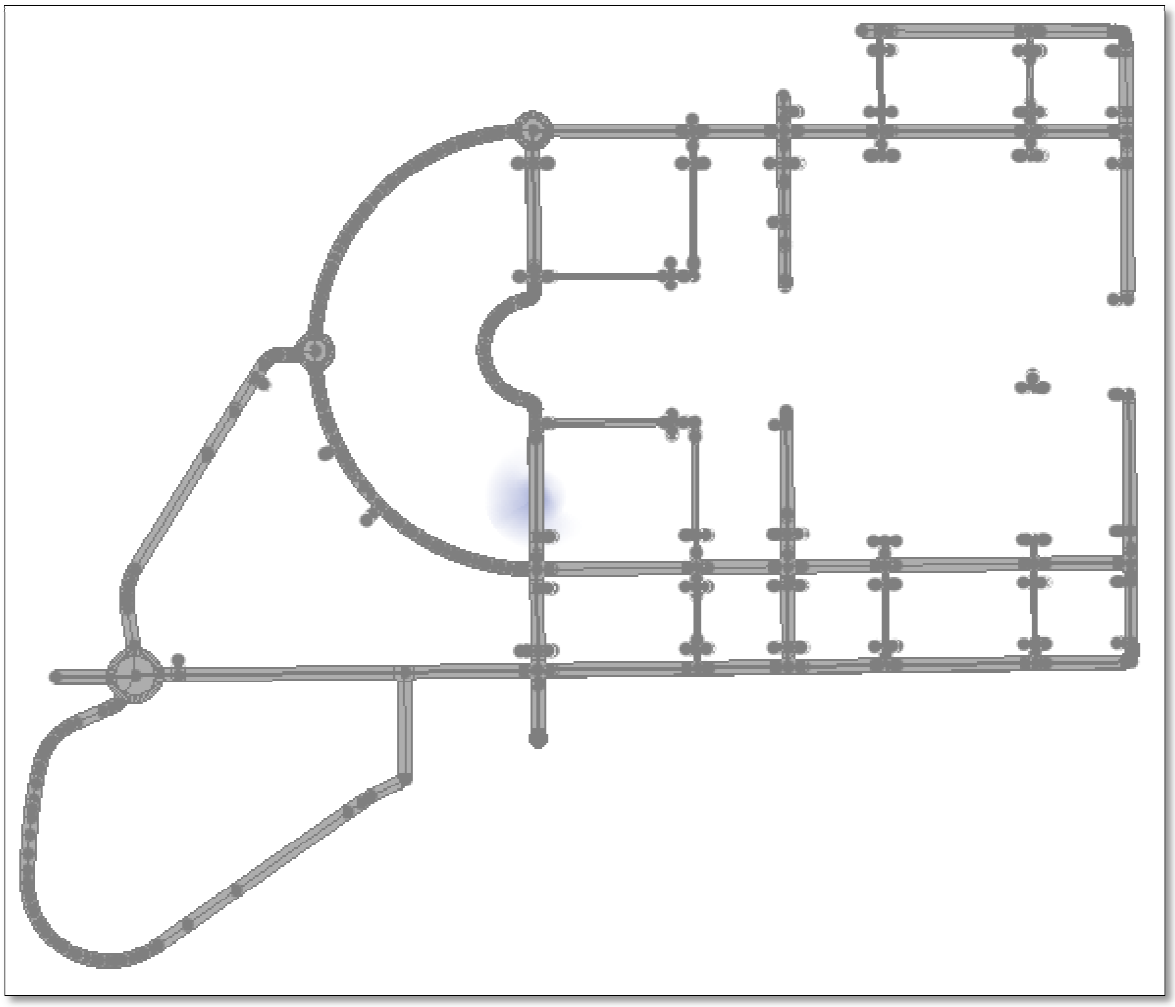

Figure 11: The completed network graph.

.

4.2.2. Assigning the Modern Streets template to the graph

Next the Modern Street Template was assigned to the newly created graph network. The template was downloaded at:

http://www.arcgis.com/home/item.html?id=6e45334d75f5423f9e44436cd0f53183 but it is also possible to find it through CityEngines help menu. The zip file contains all the files needed to assign textures and objects to the street network. It has the standard folder structure and was extracted to the projects root folder. The whole street network was selected and in the inspector window the ModernStreet.cga rule files was assigned to it. The start rule Street had to be manually assigned to the network since no start rule is defined for the rule file. Before generating the streets the rule attributes for the network where set to the following values:

```
Model Options
     Level_of_detail          :    Low (optional)

Street Layout
     Nbr_of_left_lanes        :    2
     Nbr_of_right_lanes       :    2
     Lane_width               :    3
     Median_width             :    2
     Lawns                    :    true
     Arrow_marking            :    true

Crosswalk
     Crosswalk_width          :    0
     Crosswalk_style          :    European (not used)

Lights
     Traffic_lights           :    true
     Lamps                    :    true
     Lamp_distance            :    30
```

```
Trees
     Tree_percentage          :    100
     Tree_distance            :    1
     Tree_max_height          :    1

Traffic Density
     Vehicles_per_km          :    10
     Bus_Percentage           :    0

Sidewalk (not used)
     People_percentage        :    0
     Props_percentage         :    0
     Sidewalk_height          :    0


Connected Attributes(calculated by CityEngine and should not
be modified)
     connectionEnd            :    -
     connectionStart          :    -
     type                     :    -
     elevation                :    -
```

Similarily the rule attributes for the minor streets should be the following:

```
Model Options
     Level_of_detail          :    Low (optional)

Street Layout
     Nbr_of_left_lanes        :    1
     Nbr_of_right_lanes       :    1
     Lane_width               :    3
     Median_width             :    0
```

```
     Lawns                    :    false
     Arrow_marking            :    true

Crosswalk
     Crosswalk_width          :    0
     Crosswalk_style          :    European (not used)

Lights
     Traffic_lights           :    false
     Lamps                    :    false
     Lamp_distance            :    0

Trees
     Tree_percentage          :    0
     Tree_distance            :    0
     Tree_max_height          :    0

Traffic Density
     Vehicles_per_km          :    10
     Bus_Percentage           :    0

Sidewalk (not used)
     People_percentage        :    0
     Props_percentage         :    0
     Sidewalk_height          :    0


Connected Attributes (calculated by CityEngine and should not
be modified)
     connectionEnd            :    -
     connectionStart          :    -
     type                     :    -
```

```
elevation                    :     -
```

Two new rules had to be added to the ModernStreets.cga rule file, one to model crosswalk over major streets (the zebra rule) and one to model crosswalks over minor streets (the red brick rule). The Modern Street template has an automatic way of generating crosswalks but they are too far off from the actual crosswalks of the UJI campus. As a result all Crosswalk_width attributes where set to 0 and the ModernStreet.cga rule file was opened for editing in the rule editor and the following rules were created:

Figure 12: Crosswalks over the major and minor streets at the UJI campus as seen in Google maps.

- **The red brick rule**

All crosswalks over minor streets are paved with red bricks instead of the standard Modern Streets template zebra crosswalk. A red brick texture image was downloaded from the internet and saved to streets folder for use with the network..

At line 167 in the modernStreet.cga rule file the texture was defined as:

```
redBrickTexture = "streets/redPavementTexture2crop.jpg"
```

At the end of the rule file a new rule was set up with the code:

```
RedPavement-->
     texture("redBrickTexture")
     tileUV(0, ~1, ~1)
```

The texture command sets the texture to the one defined at line 167; the tileUV operation is defined by three integers. The first one instruct the machine to use color map, ~1 and ~1 instructs the machine to use more or less the original texture x and y sizes and tile them. This rule was then set to all minor road crosswalks that where framed in part 4.2.1.

- **The zebra rule**

The zebra crosswalk is automatically generated by the Modern Streets template for the major roads but does not fit to the real situation at UJI as result it was decided to manually create a rule for it.

In this rule the zebra walkway texture from the Modern Streets template was used, found in assets/streets/crosswalk.png. As before the texture was defined now in line 168 as:

```
modifiedZebraCrosswalk = "assets/streets/crosswalk.png"

WalkingPath -->
          texture("modifiedZebraCrosswalk ")
          alignScopeToGeometry(zUp, 0, longest)
```

In this case the texture was not tiled since the whole image had to be used; the scope of the texture was set to align itself with the objects longest side for correct interpretation. No further manipulation was needed since the walking path always has the shape of a rectangle. The crosswalks framed for the major roads where then selected and the zebra rule assigned to them.

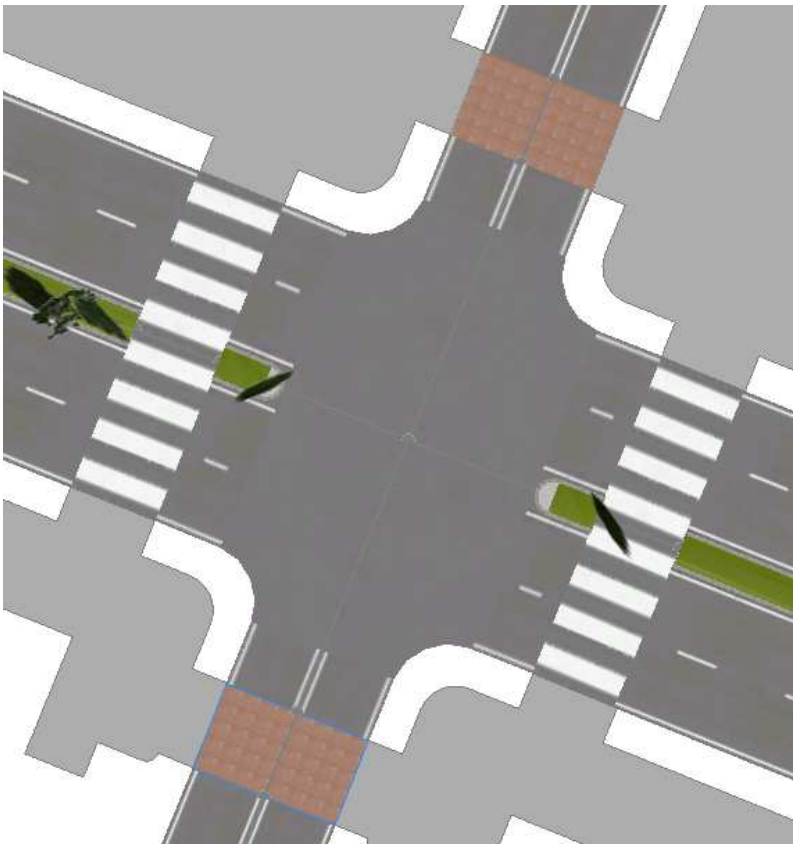

Figure 13: The major and minor road crosswalks after the new rules were assigned to them.

4.2.3. Creating a cga rule file for the LandscapeArea layer

A CGA (computer generated architecture) file is text file used to declare shape and split grammar rules for use in CityEngine. This chapter will describe in details how to CGA rule file was created for the LandscapeArea polygon layer. The LandscapeArea layer serves this purpose well since it has both simple and complex shapes that need to be modeled. The shapes are the following:

- Grass (simple texture rule)
- Gravel (simple texture rule)
- Sports Turf (simple texture)
- Ivy / Groundcover (Complex object)
- Planter (Complex object)

From the file menu a new CGA rule was created and given the name facilitiesStreet. A good practice is to divide the rule file into areas where each code type belongs to based on their functionalities. A typical CGA rule file has attributes, constants, functions textures, assets and rules. The facilitiesStreet file has attributes, textures and rules organized in the following way:

```
/**
 * File:    facilitiesStreet.cga
 * Created: 24 Jan 2013 21:28:56 GMT
 * Author:  Kiddi
 */

version "2012.1"

######################Attributes#########################
Here we define our attributes
######################Textures###########################
Here we define our textures
######################Rules##############################
Here we define our rules
```

In order to refer to the object attributes they had to be declared in the same manner as rule attributes. The object attributes where declared as an empty string since they already had an attribute as follows:

```
attr SURFTYPE = ""
```

By declaring the SURFTYPE attribute which classifies the polygon types it becomes usable within the rule file. Another attribute had to be declared but this time a rule attribute that set the sidewalk height above the streets:

```
attr Sidewalk_height = 0.2
```

The layer has the polygon type planter it is a big tree pot that surrounds the streets on the campus area. ViscaUJI has a tree layer but no trees are in the database for these trees so they had to be added to the planter. The trees were borrowed from the Modern Street template and added to the planter. They are Elm trees and need to have a max height and a distance rule attributes:

```
attr Elm_max_height = 5
attr Planter_tree_distance = rand(1,3)
```

The attributes where then categorized for the purpose of easy reading in the inspector by adding the following code before the attributes where declared:

```
@Group("Name of category")
```

For further visualization a code was added for mouse over messages that where displayed once the mouse sits over the attribute in the inspector with the code:

```
@Description("text")
```

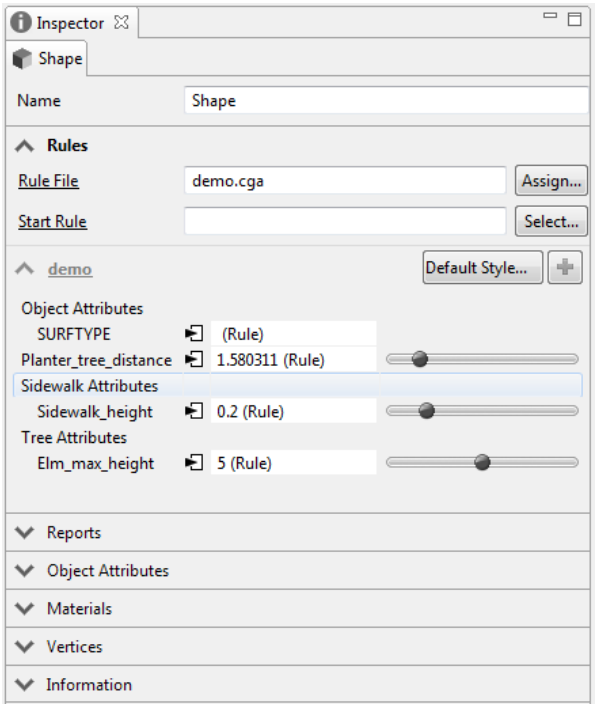

Figure 14: Screenshot of the attributes after categorisation in the inspector tab.

The next part of the rule file is the definition of textures to be assigned to the LandscapeArea and it has three types:

- Grass
- Gravel
- Curb

As before the textures where downloaded from the internet and they saved to the assets project folder. The textures where defined as follows:

```
grass_texture       = "assets/textures/grassTiles.jpg"
gravel_texture      = "assets/textures/gravel.png"
curb_texture        = "assets/streets/curb.png"
```

Figure 15: Grass texture.

Figure 17: Gravel texture.

Figure 16: Curb texture.

The next part of the rule is defines all assets used in it in much the same way as we as the textures where defined. The assets used where 3d models of trees that came with the Modern Street Template in the .obj format. There are several type of trees in the Modern Street Template and they were randomly generated by using the fileRandom function built into CityEngine:

```
Elm_Tree  = fileRandom("assets/trees/alleyTree_*_v1.obj")
```

In most cases the assets where adjusted later on to whatever context they were used. Rules were created later in the project to further define the behavior of the trees in the planter rule.

The next step was to create the actual rules that control the transformation of the shapes. The decision was made to create a so called start rule that was applied to each shape. The idea behind the start rule is to let CityEngine automatically identify the shape and assign it a

secondary rule which defines its behavior. The secondary rules vary considerably depending on the object they are intended to model so it is important that the start rule identifies the object correctly. The start checks the SURFTYPE attribute and assigns it the correct secondary rule for the shape; this is achieved with a simple switch loop. The start rule is declared with the `@StartRule` function followed by its name and the symbol `-->`

Figure 18: The tree obj model

```
@StartRule
StartRule -->
```

The switch loop:

```
Case SURFUSE == "Grass" :
grassRule
Case SURFUSE == "Gravel" :
gravelRule
Case SURFUSE == "Sports Turf" :
grassRule
Case SURFUSE == "Planter" :
planter
Case SURFUSE == "Ivy / Groundcover" :
planter
else : NIL
```

There are three secondary rules presented here, two of them are simple texture rules but the planer rule is a little more complicated. The following section explains in detail each operation of these three rules.

In the course of this description if an operation has been covered previously it will not be explained again but assumed that the reader is already familiar with it.

- **The texture rules; grassRule and gravelRule**

```
Grass -->
```

At first the up axis of the scope is set to z and the scope aligned to the shapes longest edge:

```
alignScopeToGeometry(zUp, 0,longest)
```

Set the scope of the projection:

```
setupProjection(0,scope.xz,scope.sx,scope.sz)
```

Project the texture space onto the object (UV)

```
projectUV(0)
```

Define which texture to use:

```
texture(grass_texture)
```

Then texture space was tiled approximately (~1 stands for more or less one to one ratio) in its original size (higher values stretch the image):

```
tileUV(0, ~1, ~1)
```

This code is a simple way of defining textures for a shape from a polygon layer and can be used exactly the same way with the gravel texture except texture used is different.

- **The planter rule**

The planter is an area surrounded by curbs and has gravel at the bottom and trees growing on the inside (see figure 19). These planters are defined as polygons in ViscaUJI and they need to be elevated into 3d space by extruding:

Figure 19: A planter at the campus grounds.

```
Planter -->
    extrude(Sidewalk_height)
```

This operation creates a 3d object from the selected shape and the height of the object is defined by the Sidewalk_height attribute set earlier. The object is defined to have faces and each face is worked on separately. The extruded 3d objects can either be split by vertices, faces or edges, here the object is split into two faces: sides and bottom using the comp(f) operation:

```
comp(f) { side : BorderNormal |
bottom : reverseNormals
BottomNormal PlanterTreesNormal }
```

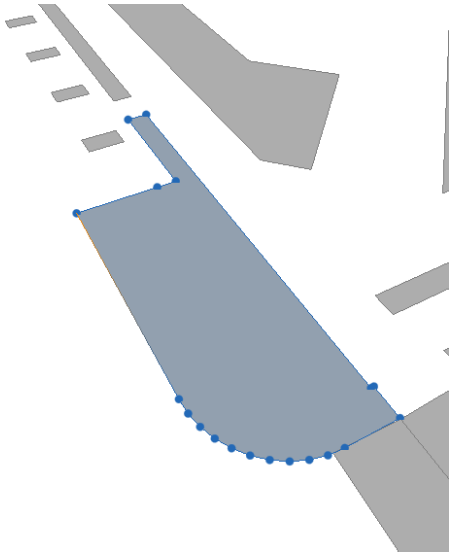

Figure 20: The polygon representation of the planter.

Since only the sides and the bottom are defined with the comp(f) function only those facades are displayed, the one remaining façade the top is discarded and not displayed. Inside the curly braces is where the facades are defined, after the colon sign any operation can be defined but usually only references to another rule is placed here as is the case with the side rule. The bottom face starts with the operation reverseNormals which is necessary since the textures and objects attached to it apply on the outside of the object by default. The BottomNormal and PlanterTreesNormal are rules that define the bottom texture and trees which will be created later.

```
setupProjection(0,scope.xz,scope.sx,scope.sz)
projectUV(0)
```

- **The Border rule**

Figure 19 shows that the sides of the planter are large concrete stones and a rule was created to model them. This rule is the one applied to the sides in the comp(f) split of the planter rule:

```
BorderNormal-->
    s('2, Sidewalk_height*4, Sidewalk_height*4)
```

The s operator sets the scope (size) of the object in the x, y and z order which is the default order used in all geometry operations. The x size is set to a value of absolute 2 which splits the border into two unit sizes (scale factor is unknown) and the absolute value instructs the program to cover all the area. The borders extrude from the sidewalk and have the same width as height so the y and z values were set as multiplication of the Sidewalk_height. In this case the multiplication factor is 4 to ensure the borders where always four times higher than the sidewalk.

```
t(-0.2, 0, -Sidewalk_height*4)
```

The t operator shifts the border a little bit to align the edges, x axis -0.2, y axis does not change and the z axis is shifted to align up with x axis in proportion with the Sidewalk_height.

```
i("builtin:cube")
```

The i operation imports the object that all these operations apply to, in this case the built in cube object.

```
Curbs2
```

At last the Curbs2 texture rule was assigned to the new shape, the Curbs2 rule is a simple texture rule with the scope set to the absolute value of 1 for x and z axis.

- **The BottomNormal rule**

The BottomNormal rule is referenced by the planter rule and is a normal texture but it was manipulated a little too raise it above the pavement texture that in some cases lies under it.

```
t(0,0,-Sidewalk_height*1.5)
```

This operation ensures that the bottom texture is always elevated 6 mm over the sidewalk texture.

- **The PlanterTreesNormal**

Inside the planter are trees that were imported objects and they needed to be distributed:

```
PlanterTreesNormal
    alignScopeToGeometry(yUp, any, longest)
    t(0,0,0.9)
    split(u,unitSpace,0){ ~ Planter_tree_distance *0.5 : NIL
    | { 0.1: Elmtree | ~ Planter_tree_distance : NIL }*
```

```
| 0.1: Elmtree | ~ Planter_tree_distance *0.5 : NIL }
```

This rule splits up the texture space (the bottom) along the longest edge and places empty space based on the Planter_tree_distance attribute at the shortest edge. Then it inserts an Elm tree at the next point by referring to the Elm tree rule and places an empty space besides it. Multiplying the operation by itself generates a repeat of the pattern that fills up the whole planter with trees.

- **The Elmtree rule**

The final piece of the puzzle is the rule that imports the elm tree object:

```
Elmtree -->
    s(0,Elm_max_height,0)
```

Sets the height of the tree to the Elm_max_height attribute the width attribute was kept at 0 in order for the tree to scale itself proportionally to the height.

```
r(0,rand(0,360),0)
```

Rotates the tree randomly around the y axis, the x and z where kept at 0 for the trees to stand upright.

```
i(ELM)
```

Imports the elm tree from previously defined assets.

Figure 21: The planter constructed in CityEngine.

There is another type of planter on the UJI campus that also contains trees. There exist a point feature with the exact location of those trees in which case a planter rule without trees was created. That planter is of the type Ivy / groundcover and it uses an identical rule as the planterTreesNormal rule but without trees.

4.2.4. Creating a cga rule file for the StreetPavement layer

The other part of the base layer is made up of the StreetPavement polygon layer. The layer shares some textures with the LandscapeArea layer so it was decided not to create a separate rule file but to add the rules to the already existing facilitiesStreet rule file. The rule was then assigned to the layer in the same manner as to the LandscapeArea layer. This was critical in order to only have to manipulate one sidewalk_height attribute, when it is manipulated in the inspector window it affects the whole base layer as a result. The objects of the StreetPavement layer are the following:

- Street (Simple texture)
- Curb (Extruded texture)
- Walking path (extruded texture)

Figure 22: The resulting planter after applying the planter rule to it.

- Sidewalk (extruded texture)
- Parking Lot (Color only)

Figures 23 to 25 show examples of the textures used, note that the pavement of UJI has both red and grey color but it was decided only to use one red tile texture. It would be beyond the scope of this project to split the pavement into red and gray areas.

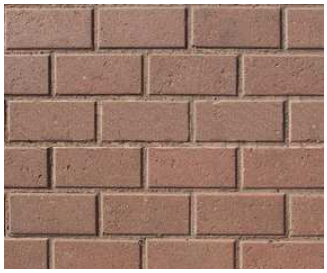

Figure 23: the texture used for the Sidewalk and curb.

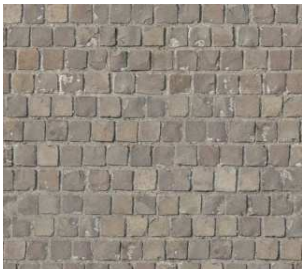

Figure 24: the texture used for the walking path.

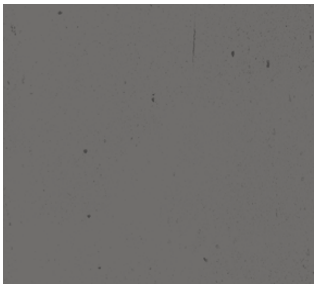

Figure 25: the texture used for the streets.

Majority of the polygons in this layer are textures and need no explanation although the sidewalk needed a little modification. The texture definition for those polygons was the following:

```
grass_texture       = "textures/grassTiles.jpg"
asphalt_texture     = "assets/streets/asphalt.png"
gray_tiles_texture  =
"textures/brick_pavement_0022_03_preview.jpg"
```

Also most of the streets polygons were generally assigned the asphalt rule from modernStreets.cga rule file (the rule was copied to the facilitiesStreet rule file) there was an exception on two polygons that represent a double zebra walkway. The big exception was the planter type in the layer that had to be assigned a planter rule similar as the one in chapter 4.2.3. The street polygons and the planter rule will be explained in the next chapter but an important difference of the StreetPavement layer is the attribute used to categorize the dataset is SURFUSE but not SURFTYPE as in the LandscapeArea layer. The SURFUSE object attribute must be declared in the facilities rule the same way as was done for the SURFTYPE object attribute. Then all the polygon types of the StreetPavement layer were added to the switch loop in the start rule. All the rules for the objects here are simple texture rules and follow the same principals as the texture rules already mentioned. However the sidewalk needed to be extruded just a little bit and needs simple border for decoration with the following rule:

```
Pavement -->
     alignScopeToGeometry(yUp, 0, longest)
     setupProjection(0,scope.xz,scope.sx,scope.sz)
     projectUV(0)
     extrude(Sidewalk_height)
     comp(f) { side :  Curbs2 | top : PavementText }
```

First the texture definition was added to texture section of the rule file:

```
red_brick_texture  =  "streets/redPavementTexture2crop.jpg"
```

The point here is to extrude the sidewalk to the actual sidewalk height and split the resulting polygon to top and sides. The top was assigned the red pavement texture above and the sides were assigned the Curbs2 texture that was used for texturing the curbs on the planter.

4.2.5. The double walking path and parking lot planter

There are two cases on the UJI campus where a walking path crosses a parking lot street polygon and the walking path has a double zebra markings (see figure 27). The double zebra had to be identified in the StreetPavement layer and its SURFUSE attribute was changed to doubleWalkingPath to separate it from the standard streets (See figure 26).

The double zebra texture was created from the zebra texture supplied with the modern street template in the Gimp photo editor and the texture definition was added to the rule file:

```
double_walking_path = "streets/doubleCrosswalk.png"
```

A new rule was created at the end of the facilitiesStreet.cga rule file for the double zebra.

The double walking path starts with the same code as the walking path rule of the street network, but since it is a polygon further functions were added:

```
setupProjection(0, scope.xy, '1, '1)
```

Instructs CityEngine to use colormap, the object scope is in the xy plane (horixontal) and the ratio between axes is absolute one to one.

```
projectUV(0)
```

Creates the final texture coordinates of the UV-set by applying the corresponding projection matrix.

rotateUV(0,90)

Rotates the UV-set 90° for perfect fit.

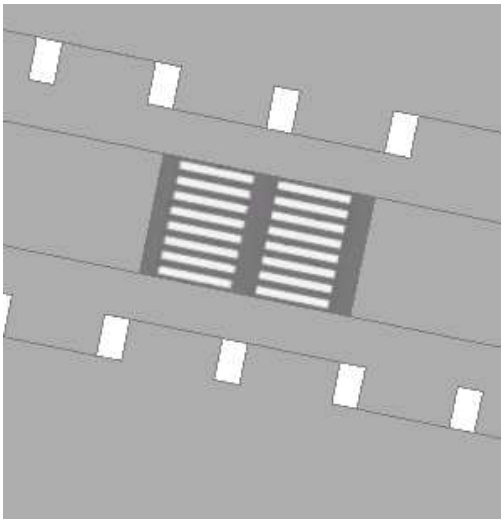

Figure 26: The double zebra polygon with the texture assigned to it.

Figure 27: The double zebra seen in Google Earth.

There are 12 polygons in the middle of all round parking spaces that represent planters that are in reality exactly like the planters in the previous example. However ViscaUJI has a layer called trees which contains the exact location of some trees including those in the parking lot planter. To generate an accurate model these tree points where used where they were available and a new rule created for the parking lot planter without the automatic generation of trees. The exact same rule was used but instead of using the split operation:

```
comp(f) { side : BorderNormal | bottom : reverseNormals BottomNormal PlanterTreesNormal }
```

the split operation:

```
comp(f) { side : BorderNormal | bottom : reverseNormals BottomNormal }
```

was used.

The 12 polygons where selected and their object attribute SURFTYPE changed from Planter to PlanterParking and the new planter rule assigned to them.

Figure 28: A street wiew from the UJI campus after generating the rules.

4.3. Indoor mapping

There is no general procedure for generating indoor 3d. maps in CityEngine and this chapter describes the process of creating a 3d indoor map for the rectorate building at the UJI campus. The goal was to create a map of the building that can be viewed from the outside to the inside using data from ViscaUJI that was created from CAD files.

4.3.1. Data preparation

In the ViscaUJI database there are two feature layers named buildingInteriorSpace and buildingFloorplanPublish. The feature layers are detailed architectural floor plans for each floor of five buildings on the campus area, each floor is categorized in the field floor. The values range from S (sub floor), 0 – 6 and C (Cubierta in Spanish or roof in English). The datasets were added to ArcGIS for modification before they could be worked on in CityEngine. The rectorate building was selected in both datasets and exported from the dataset as a two shapefiles for each dataset. When the Rectorate was exported successfully from building interiorSpaces it needed no further editing and was ready to be imported into CityEngine, the buildingFloorplanPublish needed more work though.

The BuildingInteriorSpace is the main dataset used to create the indoor map but since it is made up of polygons that represent the indoor spaces it cannot represent the exterior of the building see figure 29 - 33. The BuildingFloorplanPublish contains data that is well suited for the creation of the exterior of the building. The dataset is a complete architectural plan for the building and contains all features of the building. The idea was to use only the exterior lines of the dataset to create a outer shell covering the indoor space polygon, to achieve that goal all the lines except the once that make up the exterior had to be deleted. Special care had to be taken not delete any of the outer columns since they are part of the outer structure. Once that was complete the datasets had to be converted to polygons in ArcGIS. A new empty polygon layer was created and added it to the ArcGIS workspace, a definition query was set up on the BuildingFloorplanPublish for each floor at a time. The data was selected for each floor and the construct polygon tool used to create the polygon and add it to the new dataset. Once a polygon shell for all the floors of the Rectorate had been created the polygon layer was imported into CityEngine.

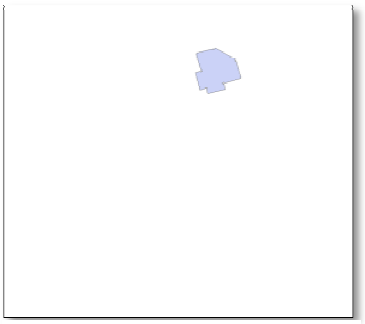

Figure 30: Indoor spaces sub floor.

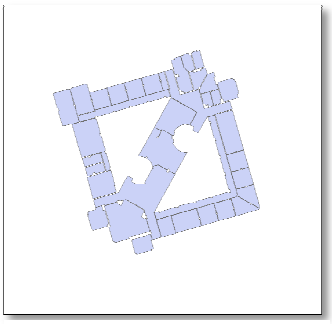

Figure 31: Indoor spaces ground floor

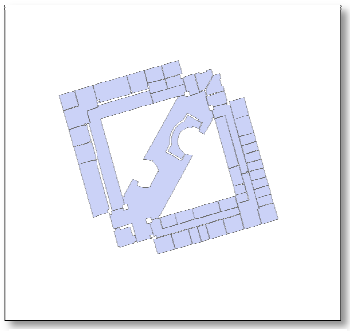

Figure 29: Indoor spaces first floor

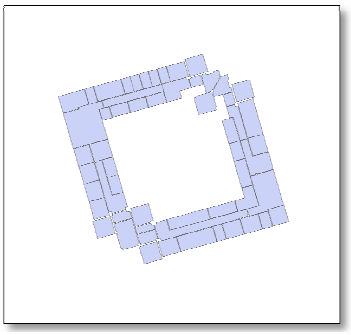

Figure 32: Indoor spaces second floor

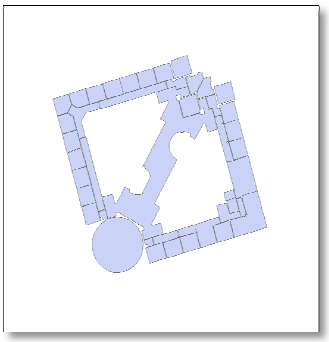

Figure 33: Indoor spaces third floor

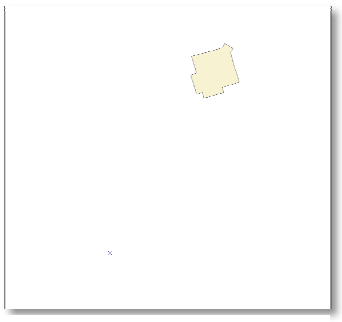

Figure 34: The sub floor shell.

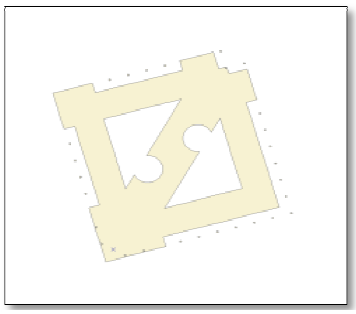

Figure 35: The ground floor shell.

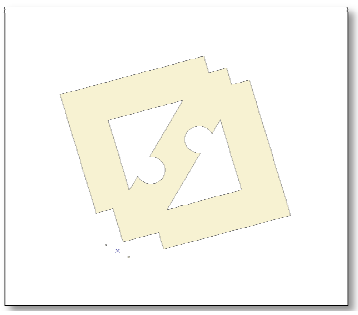

Figure 36: The first floor shell.

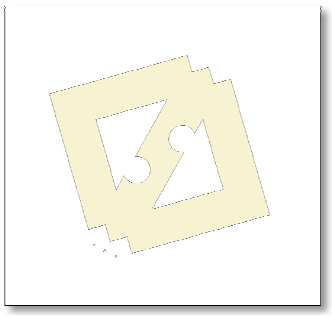

Figure 37: The second floor shell.

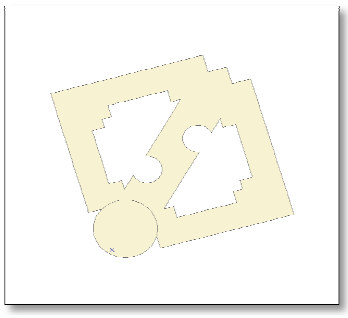

Figure 38: The third floor shell.

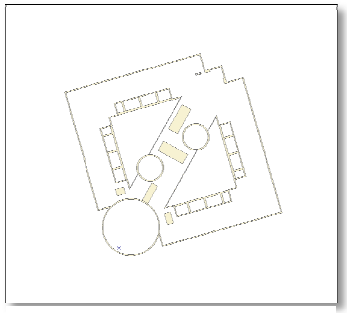

Figure 39: The roof shell.

4.4.1. The outer shell

Once the data had been imported to CityEngine a new rule file was setup in the same manner as was done in the previous examples. As before a switch loop was used for a start rule which identified each floor, extruded them and stacked them on top of each other with the t function.

The attribute used for identifying the floors is the FLOOR object attribute and had to be declared as an empty string. A rule attribute was created to control the height of the floors with name floorHeight and its default value set to 4. The basic structure of the switch loop is the following:

```
case FLOOR == "S" : extrude(-floorHeight)
t(0, -4, 0)
case FLOOR == "0" : extrude(floorHeight)
t(0, 0, 0)
case FLOOR == "1" : extrude(floorHeight)
t(0, floorHeight, 0)
case FLOOR == "2" : extrude(floorHeight)
t(0, floorHeight*2, 0)
case FLOOR == "3" : extrude(floorHeight)
t(0, floorHeight*3, 0)
case FLOOR == "roof" : extrude(floorHeight/2)
t(0, floorHeight*4, 0)

else : NIL
```

Texturing the shell was a major challenge since CityEngine does not support circles or arcs. The front of the Rectorate has on the top floor a large circular feature (see figures 39 and 40) that had to be created. In the ViscaUJI that feature is made by an arc tool that creates the arc from two endpoints, in CityEngine it becomes a straight line between those two points. The solution was to go back to ArcGIS and add multiple vertices to the arc that where spaced so close together that they simulated the arc, this method made it extremely complex to texture the arc. Each straight line of the circle is identified as a face so to split the Rectorate into faces (as is usually done with rectangular buildings) result is at least three types of faces on

the circle where there is only supposed to be one. The same happens with the columns but they are ignored here and we will focus on the big circle and the same method can be used for them. The comp(f) split function can be used for the sub, ground, first and second floors as follows:

```
comp(f) { world.north : wallLong | world.south : wallLong | world.east : wallLong | world.west : wallLong  }
```

This code split the sides of each floor depending on which direction they face and assigned them the wallLong texture rule.

What had to be done for the third floor on the other hand was a different approach, the floor was split by faces but instead of indentifying them by direction each face is assigned a number by CityEngine and assigned to it. Each line between vertices is in fact a face and in this case over 60 faces are identified just for the third floor and they all had to be assigned textures:

```
comp(f) {
      2 : tile |
      3 : wallLong |
      4 : wallLong |
      5 : tile |
      6 : tile |
      7 : tile |
      8 : tile …
      }
```

Figure 38: The Rectorate.

Figure 39: Model of the rectorate

4.4.2. The interior

The same principles outer shell was followed for the generation of indoor spaces. The BuildingInteriorSpace layer is divided into floors in the same manner as buildingFloorplanPublish layer so a switch loop was created as a start rule. As usually few things had to be inserted to the rule file before the start rule. The interior map was decorated with people models taken from the modern streets template, the following assets where therefore added to the assets part of the rule file:

```
peopleAsset =
fileRandom("assets/people_by_lowpolygon3d_com/*.obj")
```

This line randomly chooses a human model from the various ones supplied with the Modern Streets template.

dirHuman = 50%: 90 **else**: -90

This line was put for later reference and it randomly rotates the models. The walls inside the rectorate are around 15 cm thick and a infinitely thin polygon line is inefficient to intemperate them. To make the interior walls thicker a the following rule was added:

```
Wall -->
        s('1, floorHeight, 0.15)
```

Sets the scope to the full extent of the polygon layer, the height was set to the floorHeight and wall thickness is set to 0.15 meters.

```
        i("builtin:cube")
```

Imports the built in cube and applies them to the polygon layer

```
        color(1,1,1)
```

The last one is optional but in this case the walls were assigned color with the value 1,1,1 which is white (default)

To add the people models to the indoor model the following rule was created:

```
Human -->
            alignScopeToAxes(y)
            t(3,0,'rand(0.1,0.6))
```

The t (translate) function is not new but here it uses the rand function to generate objects randomly inside the object scope in this case on the y-axis.

```
            s(0,rand(1.7,1.9),0)
            r(0,dirHuman,0)
```

The r (rotate) function is already known but here it uses the dirHuman attribute to rotate the models randomly around the y axis.

```
            i(peopleAsset)
```

The last part is sets up a switch loop for the different floors that controls the extrusion and splits the shapes into faces:

Lot -->

case *FLOOR* == "IS" : extrude(-*floorHeight*)

comp(f) {side : Wall}

t(0, 0, 0)

case *FLOOR* == "I0" : extrude(*floorHeight*)
comp(f) {side : Wall | bottom : Human}
t(0, 0, 0)

case *FLOOR* == "I1" :
t(0, *floorHeight*, 0)
extrude(*floorHeight*)
comp(f) {side : Wall | bottom : Human}

case *FLOOR* == "I2" :
t(0, 2* *floorHeight*, 0)
extrude(*floorHeight*)
comp(f) {side : Wall | bottom : Human}

case *FLOOR* == "I3" :
t(0, 3* *floorHeight*, 0)
extrude(*floorHeight*)
comp(f) {side : Wall | bottom : Human}

All aspect of the switch loop have previously been covered and need no further explanation.

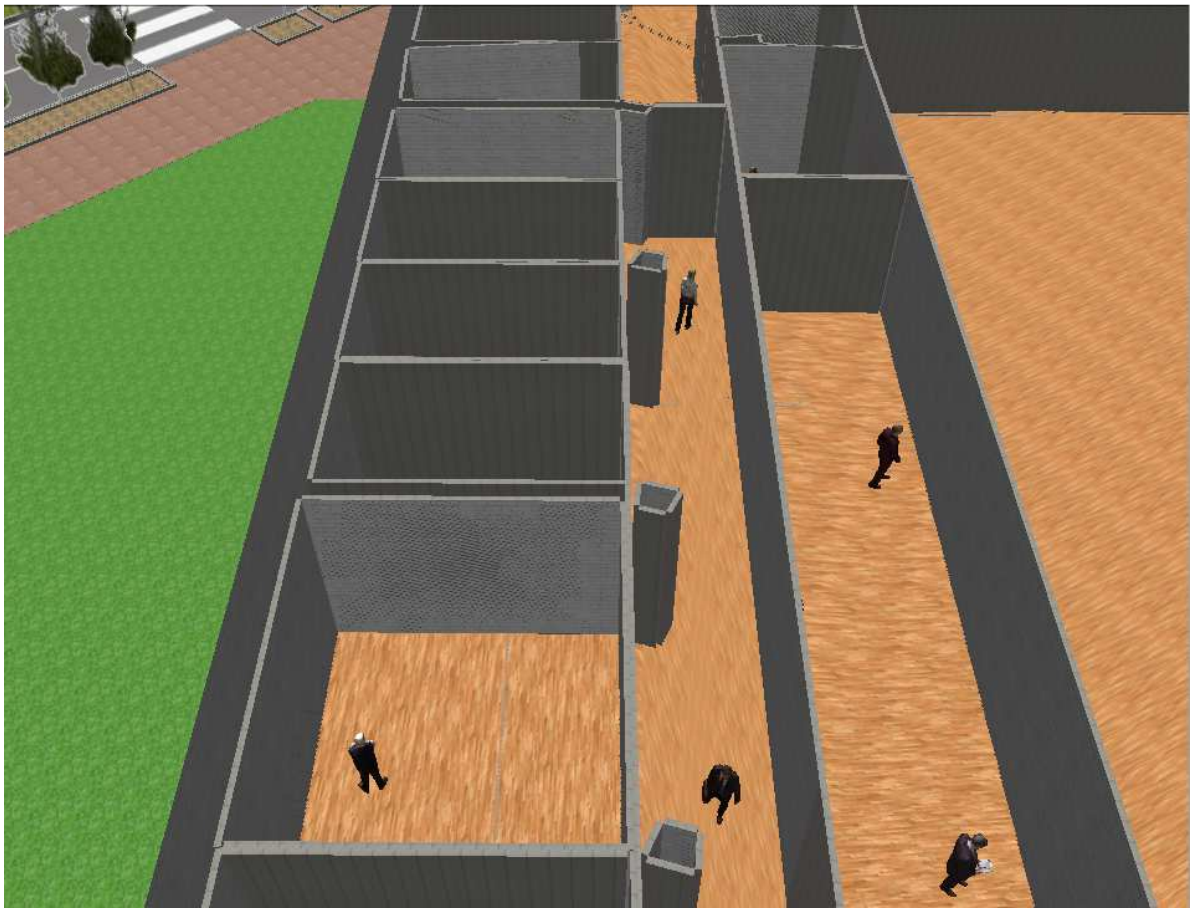

Figure 40: Part of the indoor map of the second floor.

5. Discussions

CityEngine is a powerful software package for creating realistic 3d models. The program is well suited for generating 3d content from conventional 2d GIS data types such as ESRI shapefiles and geodatabases. The following two chapters address separately the strong and weak points of the software suite from a geographers and the GIS users perspective:

5.1.1. Strong points

One of CityEngines main strong features is the ability to generate large 3d models of cities on the fly. If the user is working with building footprints of building that contain the building heights in an attribute table a low level model of large city can be generated in a few minutes with decent a computer. Such a model could for example instantly be used for skyline analysis. The user also has full control over which parts of the city he wants to add details either by using the manual the editing tools or rule based. The procedural modeling with cga shape grammar rules needs a minimal knowledge of programming and is good starting point for geographers who are more and more being forced to learn scripting languages in their use of GIS. The option of defining building parameters through attributes and modify them in the navigator resulting in instant visibility is a clever way of making programming visual.
The user interface is rich and has many good editors one of them is the façade wizard that allow you to edit a façade visually and generate from your edits a rule to apply to facades.
There is also a feature of CityEngine that is called crop image tool, although not used in the generation of the streets it is a convenient way to edit texture and facade images without having to switch to dedicated image processing software. The built in shapes of the program are vital for the generation of indoor walls, curbs, railing and other smaller details of models and hopefully they will increase in future versions.

5.1.2. Weak points

CityEngine has many features that still need to be evolved further for the software to fully utilize its potentials. One big issue with the program is how it interprets arcs and circles since they are not compatible with the way the geodatabase handles these objects. The idea behind city engine is that the user can import existing 2d data into it and generate cities. In many

cases more recent geodatabases contain objects that made up of arcs which in turn are generated from only endpoints (ArcGIS Resources, 2013a). CityEngine however does not have this function and when such a curve is imported that curve is generated as a straight line between the two endpoints. The way around this problem is to add points to the curve until with such a small space between them that they resemble a curve and can be very time consuming. Another issue on related to shapes is in regards to the manual editing tool which is missing a feature to create round shapes. As with the arcs if you want to add for example a column footprint for extrusion that has to done with straight lines in tight formation until they resemble a circle. The manual editing too is also missing other vital components such as a measuring tool to verify the dimensions of the designs. Snapping is automatic when editing manually and should be an option as it is in ArcGIS especially since it is not yet possible to create arcs and points are needed to be inserted in rapid succession. Full support to all projected coordinate system is still missing. Recently web projections have been redefined to WGS 1984 Web Mercator Auxiliary Sphere with the WKID is defined as 3857. The ViscaUJI geodatabase uses this projection also but it has not been included into CityEngine which is a big flaw since datasets are increasingly being made for the web. Although it is possible to select objects by attributes using python scripting it would be a nice feature to able to set up selection query similar to once ArcGIS has.

CityEngine has one flaw in regards to its topology, once polygons that contain holes in them are imported the software does not allow a whole in them and fills them up. This can become a problem when using CityEngine to define textures to the ground on objects such as streets and pavement since the software does not natively support layering. The workaround is to cut all polygons that contain holes in two or more segments if possible and can be very time consuming and might spoil the data. CityEngine might also benefit from proper layering such as is common in GIS software. If one layer is placed on top of another they start mixing together and the result is blurred double textures. The workaround has been to translate the top layer slightly with t function but that can become very complicated with many features and rules. CityEngine would benefit hugely if users could work on datasets natively, currently the user imports the data as a new data image. This can be a flaw if in the process of generating a 3d model some features are changed on the data they will not be changed within the database. This might create an incompatibility issues between the two datasets that are supposed to represent the same features.

One of the critical issues regarding CityEngine is at the cutting edge of 3d GIS and that is the way CityEngine currently exports the data into a File Geodatabase. There are no obvious solutions to it but the problem is that the exported data is exported to a new Geodatabase and cannot be added to an existing one unless through ArcCatalog afterwards. There are some interesting issues regarding attributes once the models have been exported. Firstly CityEngine exports the data as layers that can be viewed in all ArgGIS program but cannot be edited. This is understandable since this relates to the data model which is incomplete but raises the question of how CityEngine will be implemented into GIS software. Will it continue to be a standalone product or will its functionalities be merged with a program such as ArcScene. The models are exported as the type multipatch which allows for triangle meshes and circles with texture points and results are quite nice looking in ArcSceene. The layers are however not editable in the traditional ArgMap software suites. That is understandable given that the data model is incomplete but it will be interesting to see where this development leads to. As maps become more and more 3d a vital component is missing in the software natively and that is an object library for example street benches, lights, trees and the rest. These objects can be extracted from the various tutorials available but should be available in much the same way as textures and labeling are available in ArcGIS.

6. Conclusion

When this project started the author had no knowledge on CityEngine other that it could be used to generate 3d city models. The objective was to create a 3d model of the University Jaume I Campus and look into the possibilities of joining it up with the Smart UJI project. The software performed well in the generation of the model itself. The program has the typical learning curve where as the author was slow in adopting the many possibilities of it but as time progressed they became somewhat of experts on its many functions. Given the six month assigned to this project and delays for one month due to licensing issues it can be said that the most of the objectives were address although some of them were not satisfactorily concluded. There exists now a model prototype of the university campus that can still be further developed down to its last detail. It has been shown that the program can be used to create nice looking indoor maps but still there are unresolved issues with indoor assets such as spiral staircases.

CityEngine is a powerful software for the creation of realistic 3d models from 2d data and was successfully used to create a model of the UJI campus. The model is not so big in geographical terms but big in megabytes and might have to be redesigned if no changes will be made to how the software handles large data. The model is too big for the CityeEngine 3d web viewer to handle, when it is exported as a whole in the 3ds format the exporter crashes. When isolated parts of the model are exported as 3ds they can be viewed in the web viewer. This is likely due to the many imported .obj files contained in the model especially tree models, another factor is the many layers that the model is generated from. The model can be exported as a file geodatabase and added to ArcGIS but due to its size it works very slowly.

The model is exported as file geodatabase layers and each layer is a collection of multipatches that can be moved in 3d space but not modified. The interesting part is that if a building is split into let´s say three parts with split grammars in CityEngine when exported into ArcSceene the same building is made up of three multipatches. A single building made up of three multipatches is not very descriptive when it comes to topological relations and is difficult to query. Further work needs to be conducted in order to join the multipatches that result from split grammars. However if the split grammars are not used and only the shape grammars on a single 2d polygon the exported building is a single multipatch and such a model is usable for topological analysis and queries. The case of the indoor map of the

rectorate demonstrates this very well where a single polygon represented a single room of the building was extruded and the top removed, the resulting multipatches could be queried as single objects.

The downside to this is the fact that when the model is exported from CityEngine all attributes are rewritten. All the original attributes that where contained in the 2d indoor map of the Rectorate where removed and the final result was an attribute table with two CityEngine generated attributes. All the original attributes had to be inserted again to each corresponding room and only then the 3d indoor map is usable within the ViscaUJI smart campus project. This occurs because CityEngine does not work natively on the ArcGIS multipatch principle and all objects are rewritten into multipatch during the export process.

Bibliography

ArcGIS Resources (2013a). *ArcGIS Help 10.1 - Creating a curve through endpoints (Endpoint Arc)* Viewed January 10, 2013 at http://resources.arcgis.com/en/help/main/10.1/index.html#/through_endpoints_Endpoint_Arc/01m70000006n000000/

ArcGIS Resouces (2013b). *ArcGIS for Local Government 10.1* . Viewed January 10, 2013 at http://resources.arcgis.com/en/help/localgovernment/10.1/index.html#/Welcome_to_ArcGIS_for_Local_Government_online_help/028s00000023000000/

ArcGIS Resouces (2009). Forum *Thread: Beta 10: Topology in ArcGlobe*. Viewed January 16, 2013 at http://forums.arcgis.com/threads/1370-Beta-10-Topology-in-ArcGlobe

ArcGIS Resources, (2012). *Example Modern Streets 2012* Viewed November 19, 2012 at http://www.arcgis.com/home/item.html?id=50e9b9b6c3ea4617b5b4a7bfd4e7f38c

Cheng, R. (2012). *Google unveils full 3D Google Earth feature.* Viewed January 12, 2013 at http://news.cnet.com/8301-1023_3-57448299-93/google-unveils-full-3d-google-earth-feature/

Coppock J.T. & Rhind D.W. (1991). The History of GIS. in Maguire D.J., Goodchild M.F., Rhind D.W. (editors), *Geographical Information Systems : Principles and Applications* (pages 21-43). Longman Scientific: Essex

Elwannas, R. (2011). 3D GIS: It's a Brave new World. Speech made at The Sixth National GIS Symposium in Saudi Arabia. Viewed at http://www.gdmc.nl/3dcadastres/literature/3Dcad_2011_04.pdf

Esri (2008). *The Multipatch Geometry Type* [ESRI whitepaper]. California: ESRI Press

Goodchild, M., F. (2003). Finding the mainstream. Keynote speech at the *Map India Conference,* New Delhi.

Handrahan, H. (2011). Games *Industry International Esri acquires CityEngine developer Procedural.* Viewed Desember 26, 2013 at http://www.gamesindustry.biz/articles/2011-07-12-esri-acquires-cityengine-developer-procedural

Hohmann, B., Havemann, S., Krispel, U., Fellner, D. (2010). A GML shape grammar for semantically enriched 3D building models. *Computers & Graphics 34(4)*, 322–334

Maren, G. V., Shephard, N., Schubiger, S. (2012). *Developing with Esri CityEngine* Viewed September 29 at http://proceedings.esri.com/library/userconf/devsummit12/papers/developing_with_esri_cityengine.pdf)

Muller, P., Wonka. P., Haegler, S., Ulmer, A., Gool, L. V., Leuven, K. U. (2006). Procedural modeling of buildings. *ACM Transactions on Graphics, 25(3)*, 614 - 623

Nielsen, A. (2007). *A Qualification of 3d Geovisualization.* PhD thesis, Aalborg University, Aalborg.

Parish, Y. I., & Müller, P. (2001). Procedural modeling of cities. *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (pp. 301-308). New York: ACM.

Stiny, G., Gips, J. (1972). Shape Grammars and the Generative Specification of Painting and Sculpture. *Information Processing 71,* 1460-1465.

Stoter, J. & Zlatanova, S. (2003). *3D GIS where are we standing?* Joint Workshop on Spatial, Temporal and Multi-Dimensional Data Modelling and Analysis, 2-3 October, Quebec city, Canada.

Tuan, A. N. G. (2013). Overview of Three-Dimensional GIS Data Models. *International Journal of Future Computer and Communication, 2(3)*, 270–274

Tobler, W., R. (1970). A Computer Movie Simulating Urban Growth in the Detroit Region. *Economic Geography, 46*, 234-240

Watson, B., Müller, P., Wonka, P., Sexton, C., Veryovka, O., Fuller, F. (2008). Procedural Urban Modeling in Practice. *Computer Graphics and Applications, 28(3)* 18-26.

Wonka, P., Wimmer, M., Sillion, F., Ribarsky, W. (2003) Instant Architecture. *Transactions on Graphics 22(3)*, 669-677

Yin, X., Wonka, P., Razdan, A. (2009). Generating 3D Building Models from Architectural Drawings: a survey. *Computer Graphics and* Applications, 9(1). 20-30

Zlatanova, S. (2009). 3D model, *Geospatial today*, 09, 40-43

Zlatanova, S., Rahman, A. A., Pilouk, M., (2002). Trends in 3D GIS development. *Journal of Geospatial Engineering, 4(2)*, 1-10

Annex A

```
/**
 * File:    facilitiesStreet.cga
 * Created: 24 Jan 2013 21:28:56 GMT
 * Author:  Kiddi
 */

version "2012.1"

######################Attributes#########################

attr SURFTYPE = ""
attr SURFUSE = ""
attr NAME = ""
attr Sidewalk_height = 0.04
attr Elm_max_height = 5
@Range("Low","High")
attr Elm_Level_of_Detail = "Low"
attr Palm_max_height = 10
const highLOD = Elm_Level_of_Detail == "High"
attr Planter_tree_distance = rand(1,3)



########################Textues############################

curb_tex            = "assets/streets/curb.png"
double_walking_path = "streets/doubleCrosswalk.png"
gravel_texture      = "assets/textures/gravel.png"
grass_texture       = "textures/grassTiles.jpg"
gray_tiles_texture  =
"textures/brick_pavement_0022_03_preview.jpg"
asphalt_texture     = "assets/streets/asphalt.png"

######################Assets##############################
ELM =
     case highLOD :
fileRandom("assets/trees/tree_eu03_11_*.obj")
     else         :
fileRandom("assets/trees/alleyTree_*_v1.obj")

PALM = "assets/trees/myPalm4.obj"


#########################Rules############################

@StartRule
Lot -->
```

```
case SURFUSE == "DoubleWalkingPath" :
t(0,0.01,0)
DoubleWalkingPath

case SURFTYPE == "Planter" :
t(0,0.01,0)
Planter

case SURFTYPE == "Gravel" :
t(0,0.01,0)
Gravel

case SURFTYPE == "Ivy / Groundcover" :
t(0,0.0001,0)
PlanterNormal

case SURFTYPE == "Grass" :
t(0,0.01,0)
Grass

case SURFTYPE == "Sports Turf" :
t(0,0.01,0)
Grass

case SURFTYPE == "PlanterParking" :
PlanterNormal

case SURFUSE == "Sidewalk" :
t(0,0.01,0)
Pavement

case SURFUSE == "Street" :
t(0,0.01,0)
Ashphalt

case SURFUSE == "Walking Path" :
GreyTiles

case SURFUSE == "WalkingPathConection" :
GreyTilesConection

case SURFUSE == "Curb" :
Pavement

case SURFUSE == "Parking Lot" :
t(0,0.01,0)
ParkingSpace
```

```
case NAME == "Elm" :
Elmtree

case NAME == "Palm" :
Palmtree


else : NIL


Curbs2 -->
          alignScopeToGeometry(zUp,0)
          setupProjection(0,scope.xy,1,'1)
          projectUV(0)
          translateUV(0,-scope.sx/2,0)
          texture(curb_tex)

DoubleWalkingPath -->
          texture(double_walking_path)
          alignScopeToGeometry(zUp, 0, world.lowest)
          setupProjection(0, scope.xy, '1.0000, '1.0000)
          projectUV(0)
          rotateUV(0,90)

PlanterNormal -->
     alignScopeToGeometry(yUp, 0, longest)
     setupProjection(0,scope.xz,scope.sx,scope.sz)
     projectUV(0)
     translateUV(0,0.31,0.62)
     extrude(Sidewalk_height)
     comp(f) { side : BorderNormal | bottom : reverseNormals
     BottomNormal }

BorderNormal-->
     s('2, Sidewalk_height*4, Sidewalk_height*4)
     t(-0.1, 0, -Sidewalk_height*4)
     i("builtin:cube")
     Curbs2

BottomNormal-->
     texture("assets/textures/gravel.png")
     tileUV(0, ~1, ~1)
     t(0,0,-Sidewalk_height*1.5)
     scaleUV(0, 2, 2)

ParkingSpace-->
     color("#BBBBBB")
```

```
Pavement -->
     alignScopeToGeometry(yUp, 0, longest)
     setupProjection(0,scope.xz,scope.sx,scope.sz)
     projectUV(0)
     translateUV(0,0.31,0.62)
     extrude(Sidewalk_height)
     comp(f) { side :  Curbs2 | top : PavementText }

PavementSides-->
     tileUV(0, 0.2, 0)
     texture(curb_tex)

PavementText-->
     tileUV(0, ~1, ~1)
     texture("streets/redPavementTexture2crop.jpg")

Grass -->
     alignScopeToGeometry(yUp, 0,0)
     setupProjection(0,scope.xz,scope.sx,scope.sz)
     projectUV(0)
     translateUV(0,0,0)
     texture(grass_texture)
     tileUV(0, ~1, ~1)

Gravel -->
     alignScopeToGeometry(yUp, 0,0)
     setupProjection(0,scope.xz,scope.sx*6,scope.sz*10)
     projectUV(0)
     translateUV(0,0.31,0.62)
     texture(gravel_texture)
     tileUV(0, ~1, ~1)

GreyTiles -->
     alignScopeToGeometry(yUp, 0,0)
     setupProjection(0,scope.xz,scope.sx*6,scope.sz*10)
     projectUV(0)
     translateUV(0,0.31,0.62)
     extrude(Sidewalk_height*0.1)
     texture(gray_tiles_texture)
     tileUV(0, ~1, ~1)


GreyTilesConection -->
     alignScopeToGeometry(yUp, 0,0)
     setupProjection(0,scope.xz,scope.sx*6,scope.sz*10)
     projectUV(0)
     translateUV(0,0.31,0.62)
     extrude(Sidewalk_height * 0.05)
```

```
     texture(gray_tiles_texture)
     tileUV(0, ~1, ~1)

Ashphalt-->
     alignScopeToGeometry(yUp, 0,0)
     setupProjection(0,scope.xz,scope.sx*6,scope.sz*10)
     projectUV(0)
     texture(asphalt_texture)

Elmtree -->
     s(0,Elm_max_height,0)
     r(0,rand(0,360),0)
     i(ELM)
     set(material.opacity,1.0)

Palmtree -->
     alignScopeToAxes(y)
     s(0,Palm_max_height,0)
     r(0,rand(0,360),0)
     i(PALM)
     set(material.opacity,1.0)

Planter -->
    alignScopeToGeometry(yUp, 0, longest)
     setupProjection(0,scope.xz,scope.sx*6,scope.sz*10)
     projectUV(0)
     extrude(Sidewalk_height)
     comp(f) { side : BorderNormal | bottom : reverseNormals
     BottomNormal PlanterTreesNormal }

PlanterTreesNormal -->
     alignScopeToGeometry(yUp, 0, longest)
     t(0,0,0.9)
     split(u,unitSpace,0){ ~ Planter_tree_distance *0.5 : NIL
          | { 0.1: Elmtree | ~ Planter_tree_distance : NIL }*
          | 0.1: Elmtree | ~ Planter_tree_distance *0.5 : NIL
}
```

Annex B

```
/**
 * File:    streetsUJI.cga
 * Created: 16 May 2009 10:22:18 GMT
 * Updated: 21 Oct 2010
 */

version "2011.1"

# ----------------------------------------
# Attributes
# ----------------------------------------

@Group("Model Options",0) @Order(1) @Range("Low","High")
attr Level_of_Detail = "Low"
const highLOD = Level_of_Detail == "High"

# ----------------------
# Street Attributes

@Description("nbr of lanes. in case both are set to 0
(default), same number of lanes in both directions")
@Group("Street Layout",1) @Order(1) @Range(0,4)
attr Nbr_of_left_lanes = 2

@Description("nbr of lanes. in case both are set to 0
(default), same number of lanes in both directions")
@Group("Street Layout",1) @Order(2) @Range(0,4)
attr Nbr_of_right_lanes = 2

@Description("avg lane width, only needed when lane numbers
are not set (= 0)")
@Group("Street Layout",1) @Order(3) @Range(3,8)
attr Lane_width = 3

@Group("Street Layout") @Order(4) @Range(0,10)
attr Median_width = 2

@Group("Street Layout") @Order(5)
attr Lawns = case p(0.5): true else: false

@Group("Street Layout") @Order(6)
attr Arrow_marking = case p(0.3): true else: false

# ----------------------
# Crosswalk Attributes
```

```
@Description("if no crosswalks should be generated, it can be
set to zero")
@Group("Crosswalk",2) @Order(1) @Range(0,10)
attr Crosswalk_width =
          case lenAlongU > 3:
          case lenAlongV > 10: 4 else: 3
          else: 0

@Group("Crosswalk",2) @Order(2) @Range("American","European")
attr Crosswalk_style = "European"


# -----------------------
# Street Furniture Attributes

@Group("Lights",3) @Order(1)
attr Traffic_lights = case p(0.8): true else: false

@Group("Lights") @Order(2)
attr Lamps = case p(0.6): true else: false

@Group("Lights") @Order(3) @Range(1,30)
attr Lamp_distance = 35

@Group("Trees",4) @Order(1) @Range(0,100)
attr Tree_percentage = 60

@Group("Trees") @Order(2) @Range(2,50)
attr Tree_distance = rand(10,20)

@Group("Trees") @Order(3) @Range(2,30)
attr Tree_max_height = 7

# -----------------------
# Vehicle Attributes

@Description("vehicles per km per lane")
@Group("Traffic Density",6) @Range(0,200) @Order(1)
attr Vehicles_per_km = 0

@Group("Traffic Density") @Range(0,100) @Order(2)
attr Bus_Percentage = 3

# -----------------------
# Sidewalk Attributes

@Group("Sidewalk",7) @Order(1) @Range(0,100)
attr People_percentage = 0
```

```
@Group("Sidewalk") @Order(2) @Range(0,100)
attr Props_percentage = 0

@Group("Sidewalk") @Order(3) @Range(0,0.4)
attr Sidewalk_height = 0.2  # height of sidewalk (and depth of
curbs)

# -----------------------
# Mapped Attributes

@Group("Connected Attributes",8) @Order(1) //@Hidden
attr connectionEnd = "STREET"      # built in value
attributes, needs to be sourced as Object (parent)

@Group("Connected Attributes") @Order(2) //@Hidden
attr connectionStart = "STREET"         # built in value
attributes, needs to be sourced as Object (parent)

@Group("Connected Attributes") @Order(3) //@Hidden
attr type = "MINOR"                # built in value
attributes, needs to be sourced as Object (parent)

@Group("Connected Attributes") @Order(4)
attr elevation = 0

# ----------------------------------------
# Functions
# ----------------------------------------

calcNbrOfLanes = rint(lenAlongV/ Lane_width )
calcLanesLeft  = rint((lenAlongV*0.5-Median_width) /
Lane_width)
calcLanesRight = rint((lenAlongV - calcLanesLeft*Lane_width -
Median_width) / Lane_width)

lenAlongU = geometry.du(0,unitSpace)         # for convenience
and readability only
lenAlongV = geometry.dv(0,unitSpace)         # for convenience
and readability only

### --------------
# connection check helper functions

# connection check depending on the mapped connection
identifier
connected(ident) =
     case ident =="CROSSING" : true
     else : false
```

```
# test if shape is a connected start
isConnectedAtStart =
     connected(connectionStart)

# test if shape is a connected end
isConnectedAtEnd =
     connected(connectionEnd)

# returns 1 if a crossing at start/end, otherwise 0. Used as
split dimension for crosswalk split
conStart =
     case isConnectedAtStart : 1 else: 0
conEnd   =
     case isConnectedAtEnd: 1 else: 0

# ----------------------------------------
# Assets
# ----------------------------------------

LOD = case highLOD: 1 else: 0

# Street Textures
concrete_tex  = "assets/streets/asphalt.png"
sidewalk_tex  = "assets/streets/asphalt_brighter.png"
crosswalk_tex = case Crosswalk_style == "American":
"assets/streets/crosswalk_us.png" else:
"assets/streets/crosswalk.png"
curb_tex      = "assets/streets/curb.png"

arrows = case Arrow_marking: "arrows" else: "stop"

getStreetTexture(type,lanes) =
     case lanes >= 4 : "assets/streets/street_4lanes_" + type
+ ".png"
     case lanes < 1  : "assets/streets/street_1lanes_" + type
+ ".png"
     else            : "assets/streets/street_" + lanes +
"lanes_" + type + ".png"

# sidewalk props
newsbox1_tex  = "assets/streets/boxes/boitenews01red.png"
newsbox2_tex  = "assets/streets/boxes/boitenews01redgris.png"
mailbox_tex   = "assets/streets/boxes/mailbox01.png"
newsbox_asset = "assets/streets/boxes/newsbox.01.lod1.obj"
mailbox_asset = "assets/streets/boxes/mailbox.01.lod" + LOD +
".obj"

# signs
```

```
signs_tex = "assets/streets/signs/varioussigns.png"
getSign =
     12% : "assets/streets/signs/handicaparking.obj"
     12% : "assets/streets/signs/noparkfireline.obj"
     12% : "assets/streets/signs/noparking.obj"
     12% : "assets/streets/signs/noparkinganytime.obj"
     12% : "assets/streets/signs/noparkthiside.obj"
     12% : "assets/streets/signs/paralelparkonly.obj"
     12% : "assets/streets/signs/reservedparking.obj"
     else : "assets/streets/signs/sign_noparkfireline.obj"

# lamps
lamp2_texasset = "assets/streets/lamps/lamp.02.parallel.lod" +
LOD + ".obj"
lamp3_texasset = "assets/streets/lamps/lamp.03.withsign.lod" +
LOD + ".obj"
lamp4_texasset =
"assets/streets/lamps/lamp.04.single.lod0.obj"
lamp5_texasset =
"assets/streets/lamps/lamp.05.withsign.lod0.obj"

# trafficlights
trafficlight1_texasset =
"assets/streets/lamps/traffic_light.01.with_walking_signs.lod"
+ LOD + ".obj"
trafficlight2_texasset =
"assets/streets/lamps/traffic_light.02.big_with_lamp_and_sign.
lod" + LOD + ".obj"
trafficlight3_texasset =
"assets/streets/lamps/traffic_light.03.4sides.obj"
bigsign1_texasset =
"assets/streets/lamps/bigsign.01.luminated.lod" + LOD + ".obj"
bigsign2_texasset =
"assets/streets/lamps/bigsign.02.with_lamp.lod1.obj"

# vegetation
tree =
     case highLOD :
fileRandom("assets/trees/tree_eu03_11_*.obj")
     else         :
fileRandom("assets/trees/alleyTree_*_v1.obj")

# -------------------
# Street Rule
# -------------------

Street -->
     alignScopeToAxes(y)
```

```
     split(u,unitSpace,0){ Crosswalk_width *conStart :
Crosswalk(-1)
                         | ~1                                :
Streetsides
                         | Crosswalk_width *conEnd           :
Crosswalk(1) }

# create left and right side of the street (per default same
nbr of lanes in both directions, but user can influence it via
attributes)
Streetsides -->
     case calcNbrOfLanes < 1.1 : Asphalt
     case Nbr_of_left_lanes == 0 && Nbr_of_right_lanes == 0:
          split(v,unitSpace,0){ ~calcLanesLeft          :
Lanes(calcNbrOfLanes,connectionEnd,0) Vehicles(0)
                                     | Median_width           :
Median
                                     | ~calcLanesRight        :
scaleUV(0,-1,-1) Lanes(calcNbrOfLanes,connectionStart,2)
Vehicles(2) }
     else:
          split(v,unitSpace,0){ ~Nbr_of_left_lanes     :
Lanes(Nbr_of_left_lanes,connectionEnd,0) Vehicles(0)
                                     | Median_width           :
Median
                                     | ~Nbr_of_right_lanes    :
scaleUV(0,-1,-1) Lanes(Nbr_of_right_lanes,connectionStart,2)
Vehicles(2) }

# Lanes
Lanes(nLanes,connectionType,dir) -->
     case connectionType == "STREET" || lenAlongU < 10:
          Asphalt("stripes",nLanes)
     case connectionType == "JUNCTION" && type == "MAJOR":
          Asphalt("stripes",nLanes)
     else:
          split(u,unitSpace,0){ ~1  :
Asphalt("stripes",nLanes)
                         |  12 : Asphalt(arrows,nLanes)
TrafficLight(dir) }

# Asphalt
# - with lanes
Asphalt(type,nLanes) -->
     case lenAlongU > 3:
          [ extrude(0.1) comp(f){all: NIL} ] // required for
occlusion check
          normalizeUV(0, uv, collectiveAllFaces)
```

```
          scaleUV(0,0.5*rint(lenAlongU/6),1)
          texture(getStreetTexture(type,nLanes))
          Asphalt.
     else:
          Asphalt

# - no lanes
Asphalt -->
     [ extrude(0.1) comp(f){all: NIL} ] // required for
occlusion check
     setupProjection(0, world.xz, 12, 9) projectUV(0)
     texture(concrete_tex)
     Asphalt.

# -------------------
# Traffic Light
# -------------------

TrafficLight(dir) -->
     case Traffic_lights :
          split(u,unitSpace,0){ ~1 : NIL
                    | 0.1: split(v,unitSpace,0){ 0.1: t(0,
Sidewalk_height ,(1-dir)*2* Sidewalk_height )
alignScopeToAxes(y) TrafficLightAsset(dir) } }
     else :
          NIL

TrafficLightAsset(dir) -->
     10% : s(0,3,0) r(0,-90+dir*90,0)
i(trafficlight1_texasset)
     30% : s(0,6,0) r(0,-90+dir*90,0) i(bigsign1_texasset)
     10% : s(0,8,0) r(0,-90+dir*90,0) i(bigsign2_texasset)
     20% : s(0,2,0) r(0,-90+dir*90,0)
i(trafficlight3_texasset)
     else: s(0,6,0) r(0,-90+dir*90,0)
i(trafficlight2_texasset)

# -------------------
# Crossing & Junction
# -------------------

Crossing -->
     alignScopeToAxes(y)
     Asphalt

Junction -->
     alignScopeToAxes(y)
     Streetsides
```

```
JunctionEntry -->
     alignScopeToAxes(y)
     Asphalt

Island -->
     NIL

# -------------------
# Crosswalk
# -------------------

# splits the street the same way as before the lanes
Crosswalk(side) -->
     case Median_width > 0 && Nbr_of_left_lanes == 0 &&
Nbr_of_right_lanes == 0:
          split(v,unitSpace,0){ ~calcLanesLeft
     : CrosswalkTex
                    | Median_width - Sidewalk_height   :
Isle(side)
                    | ~calcLanesRight                  :
CrosswalkTex }
     case Median_width > 0:
          split(v,unitSpace,0){ ~ Nbr_of_left_lanes
               : CrosswalkTex
                    | Median_width - Sidewalk_height   :
Isle(side)
                    | ~ Nbr_of_right_lanes
     : CrosswalkTex }
     else :
          CrosswalkTex

# texture the shape
CrosswalkTex -->
          setupProjection(0,scope.xz,'1,10) projectUV(0)
          normalizeUV(0, uv, collectiveAllFaces)
          scaleUV(0,1,ceil(lenAlongV)/8)
          texture(crosswalk_tex)

# the isle is a small extrusion plus a round median-end is
added
Isle(side) -->
     alignScopeToGeometry(zUp, 0, 0)
     MedianWithCurb( Sidewalk_height *0.2, true)
     RoundMedian(side)


# the median end is translated and splitted according to the
side parameter
```

```
RoundMedian(side) -->
     case side == 1:
          t('1,0,0) s( Median_width , Median_width ,'1)
center(y)
          split(u,unitSpace,0){ Sidewalk_height  :
SolidCurbsNormal
                      | ~1                    :
MedianWithCurbNormal( Sidewalk_height )
                      | Median_width *0.5 : RoundCurbs(0) }
     else:
          s( Median_width , Median_width ,'1) t('-1,0,0)
center(y)
          split(u,unitSpace,0){ Median_width *0.5 :
RoundCurbs(2)
                      | ~1                    :
MedianWithCurbNormal( Sidewalk_height )
                      | Sidewalk_height  : SolidCurbsNormal }

# and insert half a cylinder
RoundCurbs(index) -->
     extrude( Sidewalk_height ) setPivot(xyz,index)
     i("streets/roundcurbs.obj")
     comp(f){ top   : split(y){ Sidewalk_height : Curbs | ~1 :
Pavement }
            | side  : Curbs }

# -------------------
# Median
# -------------------

Median -->
     MedianWithCurbs
     alignScopeToAxes(y)
     CenterLamps CenterTrees

# adds curbs on all sides
MedianWithCurbs -->
     case Crosswalk_width > 0 || Median_width < 1:
          split(u,unitSpace,0){ Sidewalk_height *conStart :
SolidCurbs
                      | ~1                              :
MedianWithCurb( Sidewalk_height , false)
                      | Sidewalk_height *conEnd   : SolidCurbs }
     else:     # in case crosswalkWidth is set to zero, we
make a round finishing
          split(u,unitSpace,0){ lenAlongV*0.5*conStart :
Asphalt RoundCurbs(2)
```

```
                    | ~1                              :
MedianWithCurb( Sidewalk_height , false)
                    | lenAlongV*0.5*conEnd   : Asphalt
RoundCurbs(0) }

MedianWithCurb(h, crosswalk) -->
     extrude(world.y,h)
     comp(f){ top  : split(v,unitSpace,0){ Sidewalk_height :
Curbs | ~1 : PavementOrLawn(crosswalk) | Sidewalk_height :
Curbs }
             | side : Curbs }

MedianWithCurbNormal(h) -->
     extrude(h)
     comp(f){ top  : split(v,unitSpace,0){ Sidewalk_height :
Curbs | ~1 : Pavement | Sidewalk_height : Curbs }
             | side : Curbs }

PavementOrLawn(crosswalk) -->
     #disabled lawn due to conn attrs on SW
     case Lawns && !crosswalk : Lawn
     else : Pavement

# distributes lamps
CenterLamps -->
     case Lamps :
          split(u,unitSpace,0){ ~ Lamp_distance :  NIL
                    | { 0.1: alignScopeToGeometry(yUp,0,0)
s(0,0,0) center(xyz) Lamp(2)
                      | ~ Lamp_distance : NIL }* }
     else : NIL

# and trees
CenterTrees -->
     case p(Tree_percentage/100) && scope.sz > 1.5:
          split(u,unitSpace,0){ ~ Tree_distance *0.5 : NIL
Lamp_distance : NIL }*
Lamp_distance *0.5 : NIL } //Commented by Kiddi to increse
trees with lamps and replaced with:
          | { 0.1: s(0,0,0) center(xyz) Tree | ~ 1 : NIL }*
          | 0.1: s(0,0,0) center(xyz) Tree | ~ 1 *0.5 : NIL }
     else : NIL

# ----------------------------------------
# Sidewalk
# ----------------------------------------

Sidewalk -->
```

```
     alignScopeToAxes(y)
     SidewalkWithCurbs
     t(0, Sidewalk_height ,0)
     SidewalkLamps SidewalkTrees SidewalkProps People

SidewalkWithCurbs -->
     extrude(world.y, Sidewalk_height )
     comp(f){ top   : split(y){ Sidewalk_height : Curbs | ~1 :
GrassPavement }
            | side : Curbs }


SidewalkWithDoubleCurbsGrass -->
     extrude(world.y, Sidewalk_height )
     comp(f){ top  : split(v,unitSpace,0){ Sidewalk_height :
Curbs | ~1 : GrassPavement | Sidewalk_height : Curbs }
             | side : Curbs }

SidewalkLamps -->
     case Lamps && lenAlongU > 5:
          split(u,unitSpace,0){ ~ Lamp_distance :  NIL
                    | { 0.1: t(0,0,scope.sz- Sidewalk_height
*2) Lamp(3) | ~ Lamp_distance : NIL }* }
     else : NIL

SidewalkTrees -->
     case lenAlongV > 3 && lenAlongU > 5:
          split(u,unitSpace,0){ ~ Tree_distance *0.5 : NIL
                    | { 0.1: Tree | ~ Tree_distance : NIL }*
                    | 0.1: Tree | ~ Tree_distance *0.5 : NIL }
     else : NIL

SidewalkProps -->
     case p(Props_percentage/100) && lenAlongU > 5:
          split(u,unitSpace,0){ Crosswalk_width *2.5+rand(-
2,2): NIL
                    | 0.1: Box | ~1: NIL | 0.1:
StreetFurniture
                    | Crosswalk_width *2.5+rand(-2,2): NIL }
          split(u,unitSpace,0){ ~ Lamp_distance *0.5 :  NIL
                    | { 0.1 : SignAsset | ~ Lamp_distance :
NIL }*
                    | 0.1 : SignAsset | ~ Lamp_distance *0.5 :
NIL  }
     else: NIL

# ---------------------
# Assets
```

```
# ---------------------

Lamp(index) -->
     alignScopeToAxes(y)      // place the Lamps vertically
     s(0,5,0)                 // set height to 5 meters
     LampAsset(index)         // since the scope's dimenstion
are zero in x and z, these are set according to the asset

LampAsset(nr) -->
     case nr == 2 : r(0,90,0) i(lamp2_texasset)
     case nr == 3 : r(0,90,0) i(lamp3_texasset)
     case nr == 4 : i(lamp4_texasset)
     else         : i(lamp5_texasset)

Tree -->
     case p(Tree_percentage/100):
          t(0,0,'0.65)                 // in the middle of
the sidewalk
          s(0,rand( Tree_max_height *0.95, Tree_max_height
),0)      r(0,rand(0,360),0)       // randdom rotate
          i(tree)                       // since the scope's
dimenstion are zero in x and z, these are set according to the
asset
          set(material.opacity,1.0)
          #set(material.opacitymap,"trees/EU03lef_v3.png")

     else : NIL

Box -->
     case p(Props_percentage/100):
          t(0,0,scope.sz- Sidewalk_height -rand(0.5,1))
          s(0,rand(0.9,1.3),0) r(scopeCenter,0,180,0)
          BoxAsset
     else : NIL

BoxAsset -->
     33%  : texture(newsbox1_tex) i(newsbox_asset)
     33%   : texture(newsbox2_tex) i(newsbox_asset)
     else : texture(mailbox_tex) i(mailbox_asset)

SignAsset -->
     case p(Props_percentage/100*0.3) :
          t(0,0,scope.sz- Sidewalk_height *2-0.25)
          s(0.5,2,0.1) r(scopeCenter,0,-90,0)
          texture(signs_tex) i(getSign)
     else : NIL

StreetFurniture -->
```

```
          s(0,0,0)
          t(0,0,rand(-.1,.4))
          i(fileRandom("assets/streetFurniture/*.obj"))

# ----------------------------------------
# Rules needed by all
# ----------------------------------------

SolidCurbs -->
     extrude(world.y, Sidewalk_height )
     comp(f){ top: alignScopeToGeometry(zUp,1) Curbs | side:
Curbs }

SolidCurbsNormal -->
     extrude( Sidewalk_height )
     comp(f){ top: alignScopeToGeometry(zUp,1) Curbs | side:
Curbs }

Curbs -->
     case scope.sx < 0.5:
          setupProjection(0,scope.xy,1,'1) projectUV(0)
          translateUV(0,-scope.sx/2,0)
          texture(curb_tex)
     else:
          setupProjection(0,scope.xy,~1.1,'1) projectUV(0)
          texture(curb_tex)



//Modified by Kiddi to control size of Laws at street sides
GrassPavement -->
     case Lawns && lenAlongV > 0 && lenAlongU > 8:
//Threshold for when lawns come in to view
          split(u,unitSpace,0){ Crosswalk_width*2: Pavement
               | split(v,unitSpace,0){'1 :
split(u,unitSpace,0){ ~20 : Gravel | { ~1:  Pavement | ~20 :
Gravel }* } //sizes of the lawns
               | ~1 : Pavement }
               | Crosswalk_width*2: Pavement}
              else :  Pavement

GravelPavement -->
          split(u,unitSpace,0){ Crosswalk_width*2: Pavement
               | split(v,unitSpace,0){'1 :
split(u,unitSpace,0){ ~20 : Gravel | { ~1:  Pavement | ~20 :
Gravel }* } //sizes of the lawns
               | ~1 : Pavement }
               | Crosswalk_width*2: Pavement}
```

```
InnerPavement -->
     split(z){~1 : Pavement | 2 : Grass}

Grass -->
     s('1,0.05,'1)
     i("/general/assets/obj/primitives/cube.notop.notex.obj")
     t(0,'-1,0)
     comp(f){bottom :
     alignScopeToGeometry(yUp, 0,0)
     setupProjection(0,scope.xz,scope.sx*6,scope.sz*10)
     projectUV(0) translateUV(0,0.31,0.62)
     set(material.colormap,"trees/EU03lef.png") Grass. }

Gravel -->
     texture("assets/textures/gravel.png")
     tileUV(0, ~0, ~2)

Lawn -->
     t(0,0,0.01)
     setupProjection(0,scope.xy,scope.sx*6,scope.sy*10)
     projectUV(0) translateUV(0,0.31,0.62)
     set(material.colormap,"trees/EU03lef.png")

Pavement -->
     setupProjection(0,scope.xy,scope.sx*6,scope.sy*10)
     projectUV(0) translateUV(0,0.31,0.62)
     set(material.colormap,"trees/EU03lef.png")

     alignScopeToGeometry(zUp, 0,0)
     Pavement(rand(10),rand(10))
     Pavement(texUOffset,texVOffset) -->
     setupProjection(0,scope.xy,12,9,texUOffset,texVOffset)
     projectUV(0)
     texture(sidewalk_tex)

# ----------------------------------------
# People
#
# Sample assets provided by lowpolygon3d.com
#
# More assets with high-res textures can be
# purchased at http://www.lowpolygon3d.com.
#
# ----------------------------------------

peopleAsset =
fileRandom("assets/people_by_lowpolygon3d_com/*.obj")
dirHuman = 50%: 90 else: -90
```

```
People -->
     case People_percentage > 0:
          50% : split(u,unitSpace,0){ { 0.1: Human |
~rand(2,5): NIL | 0.1: Human | ~rand(2,5): NIL }* | 0.1: Human
}          # could be distributed better...
          else: split(u,unitSpace,0){ { 0.1: Human |
~rand(0.5,5.5): NIL | 0.1: Human | ~rand(0.5,5.5): NIL }* |
0.1: Human }  # could be distributed better...
     else:
          NIL

Human -->
     case (scope.sz < 2 && p(People_percentage/100*0.3))
               || (scope.sz >= 2 && p(People_percentage/100)):
          t(0,0,'rand(0.1,0.6))
          s(0,rand(1.7,1.9),0) r(0,dirHuman,0)
          i(peopleAsset)
     else:
          NIL

# -----------------------------------------
# Vehicles
#
# Sample assets provided by lowpolygon3d.com
#
# More assets with high-res textures can be
# purchased at http://www.lowpolygon3d.com.
#
# -----------------------------------------

vehicleAsset(type) =
fileRandom("assets/vehicles_by_lowpolygon3d_com/"+type+"/*.obj
")

const vehiclesProb = ( Vehicles_per_km *minCarDistance)/1000
const minCarDistance = 6

Vehicles(dir) -->
     case vehiclesProb > 0:
          split(v,unitSpace,0){ ~ Lane_width :
VehiclesOnLane(dir) }*
     else:
          NIL

VehiclesOnLane(dir) -->
     case lenAlongU > 10 && p(Bus_Percentage/100):
```

```
          split(u,unitSpace,0){ ~1: VehiclesOnLane(dir) |
(rand(15,25)): Vehicle(dir,"bus") }
     case lenAlongU > 5:
          split(u,unitSpace,0){ ~1: VehiclesOnLane(dir) |
(rand(minCarDistance,15)): Vehicle(dir,"car") }
     else:
          NIL

Vehicle(dir,type) -->
     case p(vehiclesProb):
          split(u,unitSpace,0){ ~1: NIL | 0.5:
alignScopeToGeometry(yUp,dir) VehicleAsset(type) | ~1: NIL }
     else:
          NIL

VehicleAsset(type) -->
     t(0,0,'rand(0.4,0.6)) s(0,0,0) r(0,90,0)
i(vehicleAsset(type))
     set(material.opacity,1.0)


//Created by Kiddi



Curbs2 -->
          alignScopeToGeometry(zUp,1)
          setupProjection(0,scope.xy,1,'1) projectUV(0)
          translateUV(0,-scope.sx/2,0)
          texture(curb_tex)


//Street facades created by Kiddi
WalkingPath -->
          texture("/Kiddi2/assets/streets/crosswalk.png")
          alignScopeToGeometry(zUp, 0, world.lowest)
          setupProjection(0, scope.xy, '1.0000, '1.0000)
          projectUV(0)
          rotateUV(0,90)

//Street facades created by Kiddi
DoubleWalkingPath -->
          texture("streets/doubleCrosswalk.png")
          alignScopeToGeometry(zUp, 0, world.lowest)
          setupProjection(0, scope.xy, '1.0000, '1.0000)
          projectUV(0)
          rotateUV(0,90)
```

```
PlanterNormal -->
    alignScopeToGeometry(yUp, 0, longest)
     setupProjection(0,scope.xz,scope.sx*6,scope.sz*10)
     projectUV(0) translateUV(0,0.31,0.62)
     reverseNormals
     extrude(-Sidewalk_height)
     comp(f) { side : BorderNormal | bottom : BottomNormal
     PlanterTreesNormal }

BorderNormal-->
     s('1, 0.05, 0.05) t(0, '0, 0)
     i("builtin:cube")
     texture("assets/streets/curb.png")
     tileUV(0, 0.2, 0)

BottomNormal-->
    texture("assets/textures/gravel.png")
     reverseNormals
     tileUV(0, ~1, ~1)

PlanterTreesNormal -->
     alignScopeToAxes(y)
     t(0.8,0,0)
     split(u,unitSpace,0){ ~ Tree_distance *0.5 : NIL
          | { 0.1: Tree | ~ Tree_distance : NIL }*
          | 0.1: Tree | ~ Tree_distance *0.5 : NIL }

PlanterParking -->
     alignScopeToGeometry(yUp, 0, longest)
     setupProjection(0,scope.xz,scope.sx*6,scope.sz*10)
     projectUV(0) translateUV(0,0.31,0.62)
     reverseNormals
     extrude(-Sidewalk_height)
     comp(f) { side : BorderNormal | bottom : BottomNormal t(-
     0.8,0,0) PlanterTreesNormal  }

ParkingSpace-->
     alignScopeToGeometry(yUp, 0,0)
     setupProjection(0,scope.xz,scope.sx*6,scope.sz*10)
     projectUV(0) translateUV(0,0.31,0.62)
     texture("streets/concrete_wall_seamless_2_by_bitandartat-
     d3izge0.png")
     tileUV(0, ~1, ~1)

RedPavement-->
     alignScopeToGeometry(yUp, 0,0)
     setupProjection(0,scope.xz,scope.sx*6,scope.sz*10)
     projectUV(0) translateUV(0,0.31,0.62)
```

```
     texture("streets/redPavementTexture2crop.jpg")
     tileUV(0, ~1, ~1)

GrassTiles -->
     alignScopeToGeometry(yUp, 0,0)
     setupProjection(0,scope.xz,scope.sx*6,scope.sz*10)
     projectUV(0) translateUV(0,0.31,0.62)
     texture("textures/grassTiles.jpg")
     tileUV(0, ~1, ~1)

GravelTexture -->
     alignScopeToGeometry(yUp, 0,0)
     setupProjection(0,scope.xz,scope.sx*6,scope.sz*10)
     projectUV(0) translateUV(0,0.31,0.62)
     texture("textures/gravel.png")
     tileUV(0, ~1, ~1)

GreyTiles -->
     alignScopeToGeometry(yUp, 0,0)
     setupProjection(0,scope.xz,scope.sx*6,scope.sz*10)
     projectUV(0) translateUV(0,0.31,0.62)
     extrude(Sidewalk_height)
     texture("textures/brick_pavement_0022_03_preview.jpg")
     tileUV(0, ~1, ~1)

GreyTilesConection -->
     alignScopeToGeometry(yUp, 0,0)
     setupProjection(0,scope.xz,scope.sx*6,scope.sz*10)
     projectUV(0) translateUV(0,0.31,0.62)
     extrude(Sidewalk_height * 0.35)
     texture("textures/brick_pavement_0022_03_preview.jpg")
     tileUV(0, ~1, ~1)
```

Annex C

```
/**
 * File:    interior.cga
 * Created: 7 Jan 2013 19:24:24 GMT
 * Author:  Kiddi
 */

version "2012.1"

TextureCaravistaBlanco =
"assets/facades_uji/CARAVISTA_BLANCO.jpg"
VenRectorate = "assets/facades_uji/VEN_RECTORATE.jpg"
TextureFloor = "assets/facades_uji/wood-floorboards-
texture.jpg"

@Group("Floor attributes")
@Description("nbr of lanes. in case both are set to 0
(default), same number of lanes in both directions")
attr floorHeight = 4

@Group("Floor attributes")
attr FLOOR = ""

@Group("People")
attr People_percentage = 100

######################Interior#######################
@StartRule
Lot -->
case FLOOR == "IS" : extrude(-floorHeight)
color("white")
comp(f) {side : Wall}
t(0, 0, 0)

case FLOOR == "I0" : extrude(floorHeight)
color("white")
comp(f) {side : Wall | bottom : Human}
t(0, 0, 0)

case FLOOR == "I1" :
t(0, 4, 0)
extrude(floorHeight)
color("white")
comp(f) {side : Wall | bottom : Human}


case FLOOR == "I2" :
t(0, 8, 0)
```

```
extrude(floorHeight)
color("white")
comp(f) {side : Wall | bottom : Human}

case FLOOR == "I3" :
t(0, 12, 0)
extrude(floorHeight)
color("white")
comp(f) {side : Wall | bottom : Human}

//Sub Floor
case FLOOR == "S" : extrude(-floorHeight)
t(0, 0, 0)
comp(f) {world.down : Floor | side : tile }

################First floor##################

case FLOOR == "0" :
extrude(floorHeight)
t(0, 0, 0)
comp(f) {world.down : Floor | side : tile }

//Columns
case FLOOR == "C0" : extrude(floorHeight)
color(0.211,0.211,0.211)

//Open space
case FLOOR == "B" : extrude(floorHeight)
comp(f) {
      0 : color(0.211,0.211,0.211) X | //Bottom
      1 : color(0.211,0.211,0.211) X  | //Top
      2 : tile  //inside wall

      }

######################Second floor######################
case FLOOR == "1" : extrude(floorHeight)
t(0, 4, 0)
comp(f) {world.down : Floor | side : tile}

//Columns
case FLOOR == "C1" : extrude(floorHeight)
t(0, 4, 0)
color(1,1,1)

######################Third floor######################
case FLOOR == "2" : extrude(floorHeight)
t(0, 8, 0)
```

```
comp(f) {world.down : Floor | side : tile}

     //Columns
case FLOOR == "C2" : extrude(floorHeight)
t(0, 8, 0)
color(1,1,1)

//Fourth floor
case FLOOR == "3" : extrude(floorHeight)
t(0, 12, 0)
comp(f) {world.down : Floor | side : tile}

case FLOOR == "C4" : extrude(floorHeight)
t(0, 12, 0)

####################Roof#######################
case FLOOR == "5" : extrude(2)
t(0, 16, 0)
comp(f) {world.north: tile | world.south: tile | world.west:
tile | world.east: tile | world.up: color(0.211,0.211,0.211) X
| world.down: tile}

case FLOOR == "RoofTop" :
t(0,16,0)
tile

case FLOOR == "GroundRoof" :
t(0,4,0)
tile

case FLOOR == "SubRoof" :
tile

else : NIL

floor0Long -->
     //split(y) {~1: tile | ~1: window | ~0.2: tile }
     split(x){~2 : tile | split(y){~2 : tile | ~2 : window |
     ~0.5 : tile} | ~2 :tile }
     projectUV(0)
     tileUV(0, ~1, ~1)

floor1Long -->
     //split(y) {~1: tile | ~1: window | ~0.2: tile }
     split(x){~3.95 : tile | split(y){~2 : tile | ~2 : window
| ~0.5 : tile} | ~5.8 :tile }
     projectUV(0)
     tileUV(0, ~1, ~1)
```

```
floor2Long -->
     //split(y) {~1: tile | ~1: window | ~0.2: tile }
     split(x){~3.95 : tile | split(y){~2 : tile | ~2 : window
     | ~0.5 : tile} | ~11 :tile }
     projectUV(0)
     tileUV(0, ~1, ~1)

     floor3Long -->
     //split(y) {~1: tile | ~1: window | ~0.2: tile }
     split(x){~9.1 : tile | split(y){~2 : tile | ~2 : window |
~0.5 : tile} | ~5.8 :tile }
     projectUV(0)
     tileUV(0, ~1, ~1)

tile -->
     texture(TextureCaravistaBlanco)
     projectUV(0)
     tileUV(1, ~1, ~1)

window -->
     texture(VenRectorate)
     projectUV(0)
     tileUV(0.5, ~1, ~1)

Floor -->
     reverseNormals
     projectUV(0)
     tileUV(0, ~2, ~2)
     //color("gray")
     texture(TextureFloor)

##########################Interior#####################

peopleAsset =
fileRandom("assets/people_by_lowpolygon3d_com/*.obj")
dirHuman = 50%: 90 else: -90


Human -->
          alignScopeToAxes(y)
          t(3,0,'rand(0.1,0.6))
          s(0,rand(1.7,1.9),0) r(0,dirHuman,0)
          i(peopleAsset)

Wall -->
     s('1, floorHeight, 0.1) t(0, '0, 0)
     i("builtin:cube")
```

```
texture("assets/streets/curb.png")
tileUV(0, 0.2, 0)
```

Annex D

```
/**
 * File:    exteriorShell.cga
 * Created: 7 Jan 2013 19:24:24 GMT
 * Author:  Kiddi
 */

version "2012.1"

TextureCaravistaBlanco =
"assets/facades_uji/CARAVISTA_BLANCO.jpg"
VenRectorate = "assets/facades_uji/VEN_RECTORATE.jpg"


attr floorHeight = 4
attr FLOOR = ""


######################Exterior#######################

Lot -->
//Sub Floor
case FLOOR == "S" : extrude(-floorHeight)
t(0, 0, 0)
comp(f) {world.north: tile | world.south: tile | world.west:
tile | world.east: tile | world.up: color(0.211,0.211,0.211)
X}

#################First floor###################

case FLOOR == "0" : extrude(floorHeight)
t(0, 0, 0)
comp(f) {
      0 : color(0.211,0.211,0.211) X | //Bottom
      1 : color(0.211,0.211,0.211) X  | //Top
      2 : tile | //inside wall
      3 : tile | //inside wall
      4 : tile | //inside Wall
      5 : tile | //inside wall
      6 : tile | //corner Wall
      7 : tile | //inside wall
      8 : tile | //long wall
      9 : floor0Long | //inside wall
     10 : tile | //corner wall
     11 : tile | //corner wall
     12 : tile | //inside wall
     13 : tile | //long wall
     14 : floor0Long | //inside wall
```

```
      15 : tile | //corner wall
      16 : tile | //corner stub
      17 : tile | //corner stub
      18 : tile | //corner stub
      19 : tile | //corner wall
      20 : tile | //corner wall
      21 : tile | //corner wall
      22 : tile | //corner stub
      23 : tile | //corner stub
      24 : tile | //corner wall
      25 : tile | //corner wall
      26 : tile | //long wall
      27 : tile | //long wall
      28 : tile | //corner wall
      29 : floor0Long |  //corner wall
      30 : floor0Long |
      31 : tile |
      32 : tile
      }
//Columns
case FLOOR == "C0" : extrude(floorHeight)
color(0.211,0.211,0.211)

//Open space
case FLOOR == "B" : extrude(floorHeight)
comp(f) {
      0 : color(0.211,0.211,0.211) X | //Bottom
      1 : color(0.211,0.211,0.211) X  | //Top
      2 : tile  //inside wall
      }

########################Second floor#########################
case FLOOR == "1" : extrude(floorHeight)
t(0, 4, 0)
comp(f) {
      2 : tile | //corner wall
      3 : floor1Long | //Long wall
      4 : floor1Long | //Long wall
      5 : tile | //?
      6 : tile |
      7 : tile |
      8 : tile |
      9 : floor1Long |
      10 : floor1Long |
      11 : tile |
      12 : tile |
      13 : tile
      }
```

```
//Columns
case FLOOR == "C1" : extrude(floorHeight)
t(0, 4, 0)
color(1,1,1)

######################Third floor#######################
case FLOOR == "2" : extrude(floorHeight)
t(0, 8, 0)
comp(f) {
     2 : tile |
     3 : floor1Long |
     4 : floor2Long |
     5 : tile |
     6 : tile |
     7 : tile |
     8 : tile |
     9 : tile |
     10 : tile |
     11 : tile |
     12 : tile |
     13 : tile |
     14 : tile |
     15 : floor3Long |
     16 : floor1Long |
     17 : tile |
     18 : tile |
     19 : tile |
     20 : tile
     }

     //Columns
case FLOOR == "C2" : extrude(floorHeight)
t(0, 8, 0)
color(1,1,1)

//Fourth floor
case FLOOR == "3" : extrude(floorHeight)
t(0, 12, 0)
comp(f) {
      2 : tile |
      3 : floor1Long |
      4 : floor1Long |
      5 : tile |
      6 : tile |
      7 : tile |
      8 : tile |
      9 : tile |
```

```
10 : tile |
11 : tile |
12 : tile |
13 : tile |
14 : tile |
15 : tile |
16 : tile |
17 : tile |
18 : tile |
19 : tile |
20 : tile |
21 : tile |
22 : tile |
23 : tile |
24 : tile |
25 : tile |
26 : tile |
27 : tile |
28 : tile |
29 : tile |
30 : tile |
31 : tile |
32 : tile |
33 : tile |
34 : tile |
35 : tile |
36 : tile |
37 : tile |
38 : tile |
39 : tile |
40 : tile |
41 : tile |
42 : tile |
43 : tile |
44 : tile |
45 : tile |
46 : tile |
47 : tile |
48 : tile |
49 : tile |
50 : tile |
51 : tile |
52 : tile |
53 : tile |
54 : tile |
55 : tile |
56 : tile |
57 : tile |
```

```
       58 : tile |
       59 : tile |
       60 : tile |
       61 : tile |
       62 : tile |
       63 : tile |
       64 : tile |
       65 : tile |
       66 : tile |
       67 : tile |
       68 : floor1Long |
       69 : floor1Long |
       70 : tile |
       71 : tile |
       72 : tile |
       73 : tile |
       74 : tile |
       75 : tile |
       76 : tile
       }

case FLOOR == "C4" : extrude(floorHeight)
t(0, 12, 0)

####################Roof#######################
case FLOOR == "5" : extrude(2)
t(0, 16, 0)
comp(f) {world.north: tile | world.south: tile | world.west:
tile | world.east: tile | world.up: color(0.211,0.211,0.211) X
| world.down: tile}

else : NIL

floor0Long -->
     //split(y) {~1: tile | ~1: window | ~0.2: tile }
     split(x){~2 : tile | split(y){~2 : tile | ~2 : window |
     ~0.5 : tile} | ~2 :tile }
     projectUV(0)
     tileUV(0, ~1, ~1)

floor1Long -->
     split(x){~3.95 : tile | split(y){~2 : tile | ~2 : window
     | ~0.5 : tile} | ~5.8 :tile }
     projectUV(0)
     tileUV(0, ~1, ~1)

floor2Long -->
```

```
     split(x){~3.95 : tile | split(y){~2 : tile | ~2 : window
     | ~0.5 : tile} | ~11 :tile }
     projectUV(0)
     tileUV(0, ~1, ~1)

     floor3Long -->
     //split(y) {~1: tile | ~1: window | ~0.2: tile }
     split(x){~9.1 : tile | split(y){~2 : tile | ~2 : window |
     ~0.5 : tile} | ~5.8 :tile }
     projectUV(0)
     tileUV(0, ~1, ~1)

tile -->
     texture(TextureCaravistaBlanco)
     projectUV(0)
     tileUV(1, ~1, ~1)

window -->
     texture(VenRectorate)
     projectUV(0)
     tileUV(0.5, ~1, ~1)

##########################Interior####################

Indoor -->
case FLOOR == "IS" : extrude(-floorHeight)
t(0, 0, 0)

case FLOOR == "I0" : extrude(floorHeight)
t(0, 0, 0)

case FLOOR == "I1" : extrude(floorHeight)
t(0, 4, 0)

case FLOOR == "I2" : extrude(floorHeight)
t(0, 8, 0)

case FLOOR == "I3" : extrude(floorHeight)
t(0, 12, 0)

else : NIL
```