Beteckning:_____

HÖGSKOLAN
I GÄVLE

Institutionen för matematik, natur- och datavetenskap

A method to generate modern city buildings with the aid of Python-scripting

Erkan Dogantimur
May 2009

Thesis, 15 Credits, C level
Computer science

Kreativ Programmering
Supervisor/Examiner: Anders Hast
Co-examiner: Stefan Seipel

A method to generate modern city buildings with the aid of Python-scripting

by

Erkan Dogantimur

Institutionen för matematik, natur- och datavetenskap
Högskolan i Gävle

S-801 76 Gävle, Sweden

Email:
Dogantimur_erkan@live.se
Nkp06edr@student.hig.se

Abstract

It takes time to model buildings in a 3D city environment, for example in a game. Time is usually something very constricted in a production stage of anything, whether it is a personal project at home, at school or more occurring; in the 3D industry. This report will bring forth a method to quickly generate detailed buildings with the help of Python scripting, integrated in Maya 2009. The script will be working with modules that will be assembled together to create a modern city type of building. A comparison will be made between this script and a couple other scripts that offer the same solution but in different ways.

keywords: 3D, modeling, modern, buildings, city, Python, script, Maya 2009

Table of Content

1 Introduction

Building city scenes in 3D can be time consuming by its nature. There are lots of buildings on a modern mega city street, and they are big. Even though they may share a generic look among them, they are still different from each other. It is possible to manually model a few buildings, duplicate them and then do small adjustments to differentiate them from one another, but it is not safe to say that this is the fastest and most convenient way of doing it.

The process of creating city scenes is something that could facilitate a lot with the help of automation. By creating a system where the computer randomly selects modules or "pieces" to assemble together into buildings, there can be a lot of time to save, and unnecessary repetitive work to spare. The time saved could instead be used for simply tweaking these buildings such as adding small decals, signs, advertisement etc. on the buildings, instead of creating the entire building manually.

Luckily, we have an easy access to scripts within 3D modeling suites today, and it is possible to create your own tools to be used within the given application.

1.1 Background

The further into development we come, the more ways are discovered on how to automate processes that are repetitive. The eyes of computer science are constantly gazing upon the world, searching for patterns which can be broken down and systemized. This allows for great development and increases our understanding of anything and everything we encounter. New time saving methods are developed, whether it be generating buildings or recognizing faces through digital images, or anything in between or even outside of that range. But for this report, let us aim in on methods to model cities in a 3D environment.

There already are techniques that can quickly generate cityscapes, such as collecting geographic data with the help of laser scanners mounted on ground vehicles or airborne vehicles. Avideh Zakhor and Christian Früh from UC Berkeley have successfully developed and used this technique [1] to map three blocks of downtown Berkley, into 3D data.

There are also some older techniques that involve modeling a city from complete pre-existing buildings [4], and there are some newer techniques that can model an entire city with minimal human input [5]. There are also techniques somewhere in between manual labor and automatic processes, such as the semi-automatic CC-Modeler [9].

There will be more information about all of these in section 3.

2 Aim of research

This research is primarily aimed at finding a method to construct detailed buildings, without using primitive objects as the basis for the construction. The method is going to be quite simple and straight forward, and will allow the users to modify the modules to customize the appearance even further. The research also compares the results of this research with the results of the related work which has been referenced.

The aim of this research is not to create a huge city with one button click. The idea is that the users only create the buildings they actually need, thus making this research more appropriate for more constricted city environments where there is no need to have every single building in high detail. For example, one cannot travel everywhere in most games, since the developers constrain the area of interest. There is no need to model buildings that the player cannot reach in the far horizon, if the game itself takes place in the middle of the city. The same thing goes for animations;

nobody models or animates the unseen. This is because of technical and time constraints, since it is not optimal to spend several thousands of polygons per building in a massive city, let alone spending time on it. That amount of detail should only be where it is needed, instead of being everywhere. This means that a lot of resources are saved, and the buildings off in the distance can be of much simpler design since all the detail will be lost when the buildings with thousands of polygons are merely represented by a few pixels anyway.

The aesthetic aim is to create mostly rectangular buildings which make up the most part of any common city. Any intricate detail generation will be put on the to-do-list, since that is not the main priority. The main priority is to create detailed buildings that do not look identical. The level of detail will for now be kept at a level where textures would still contribute to the level of detail instead of being obsolete because the geometry itself is so highly detailed, that there is no need for bump mapping and so on. Texturing however is also a second priority and is placed on the to-do-list.

An appropriate application of the results of this research could be in short films where the artist needs to create a city street. In films, the storyboard already determines the camera angles, which means that the artist already knows what he needs in his city scene. Another application could be in games. Since many of today's games can load the levels dynamically as they go, they may not be as limited by polygon count as 3D scenes are in applications like Maya and such. Computer simulations where buildings are involved could also be an application for this research.

2.1 Problem definition

The biggest problem in modeling a city is obviously the time required. It would take too much time to manually model every building, and not many individuals would consider modeling an entire city by hand. This leads to duplication, and lots of it, which also leads to the next problem; repetitive looks.

It is not very funny to see the same block being repeated over and over again while driving down the street in some computer game or animation. There must be variation between the buildings in order to make the user focus on the game or animation itself, rather than the fact that these things are duplicated and repeated.

There is another problem with time, not only concerning the modeling of buildings but also this very research. This research, including the writing of this report is limited by 10 weeks in total. Final presentations occur at the last week, which brings down the effective time down to 9 weeks. Also, my programming skills are of a basic level gained from assignments in classes, which will keep the challenge very high throughout the entire project.

2.2 Questions to solve

With everything that has been said in mind, a few questions pop up on how to achieve the goal of this research.

Considering the main goal is to create buildings that have a fairly high amount of detail without looking duplicated or identical, which method is appropriate to reach that goal?

Why work in Python when Maya also offers MEL scripting?

How do users interact with the script?

Even if mass producing buildings is not a goal at this stage, how big of a city is possible to create with this method on an average computer?

2.3 Solving the problems

A method is needed to quickly create buildings while offering control over the appearance. In addition to the general shape of the building, there needs to be options to distinctively alter the features of the building. The most crucial factors are obviously time, but also variation. One of the aims is to avoid identical buildings at all costs.

2.4 Method to use

The solution will be based on scripting in Python. Python is a programming language, and it is integrated in Maya which gives the users the ability to make their own scripts in addition to Maya's own language which is called MEL (Maya Embedded Language). The reason why Python is chosen over Maya's own MEL is because Python is easier to work with [6], and offers more compatibility with other applications if need be, which MEL does not. For example, the open source 3D suite Blender [7] also has Python integrated.

The script will use small 3D modules to assemble into buildings instead of constructing the buildings from primitive objects. There will be many modules for the script to randomly choose from, so that no building looks exactly the same as another. The modules will be 2 Maya units wide and 1 Maya unit high (2X, 1Y), for the sake of consistency. The fact that the modules are of the same dimensions will aid in the scripting process and eliminate some early problem encounters, for example the placement of modules.

The script will give the user control over basic inputs like width, depth and height of a building, via a graphical user interface (although a simple one). There will also be controls for various details like ventilation shafts and antennas on the roof, and such. An option to bevel the corners will also be added in order to break away from the cubic appearance that will be apparent if every building has sharp 90-degree corners.

2.5 Demarcation

As mentioned already, the computer will not generate the buildings entirely from nothing without any user input. Quite the opposite, the computer will generate the buildings from modules which will be modeled with a lot of artistic freedom to infuse some "life" into the buildings, and the script will need the user's input to generate the buildings. If the script would indeed generate the buildings completely automatically with no user input, there would be a multitude of different appearances the buildings could have. Due to the time constraints on this research, most of the time would probably be spent on telling the computer how *not* to make the buildings, rather than *how* to. Therefore, it seems to be the better choice to simply create modules so that the appearance and style of the building is already determined. This also allows for interesting expansion packs in the future of the project, where new modules could be made in different styles, to create different buildings, like old brick houses instead of modern skyscrapers and so on.

3 Related work

3.1 Clone Attack!

In a research paper written by a group of researchers at Trinity Collage Dublin [2], the method of how to make a large crowd is presented. This is interesting and to a certain degree related to this paper. The knowledge gained from how to make large

crowds in an efficient and time saving way, could very well have positive effects on creating buildings and city scenes. Many games suffer from the fact that some or many of the NPCs (NPC stands for Non-Player Controllable) are duplicates of each other, which has a negative effect on how the user perceives the game. It is hard to submerge into the game when one constantly discovers patterns that are non-occurring in reality, such as human clones. Although buildings may be a lot more forgiving in that sense, it is still not common to regularly spot exact identical duplicates of buildings while walking down a city street.

In the previously mentioned research paper, 20 different human models are used to create a crowd. This is accomplished by duplicating the models and applying changes to alter them from each other. The most important features to alter, in order to avoid clone detection, are colors, animation and clothes/accessories. Animation is irrelevant since the buildings are obviously not going to be moving, but colors and accessories/detail is of relevance. The most important thing to learn from the mentioned research is the conclusion that the human eye is most sensitive to things that have the same shape or colors, and less sensitive to things that move exactly in the same way. Since the buildings are stationary, it is extra important to make sure that no building looks exactly the same as another. Texturing has not been implemented in this project due to time constraints, but the variation of detail on the buildings has been.

3.2 MEL Cityscape Generator

Another related work is Richard Sun's MEL Cityscape Generator [3]. This is a MEL script which works for Maya exclusively. This script adds a lot of extra modeling features that speed up many modeling processes in Maya, but the main raison d'être of the script is its ability to generate massive cities. These kinds of scripts tend to create the buildings with a low amount of detail since there are hundreds and hundreds of buildings in a big city scene. If they had a higher level of detail, it would be extremely heavy to process. Regardless though, a close-up shot from a street inside the city would not look very detailed and realistic with this script, since it uses 3D primitives to build everything from. A stretched out cube becomes a skyscraper, which is then combined with other cubes, cylinders or spheres to add further detail to the building, and so on.

Richard Sun's work shows that it is possible to generate massive cityscapes with the aid of scripts, but in order to make something more detailed and more controlled, pre-modeled modules will be used instead.

3.3 CC-Modeler: a topology generator for 3-D city models

CC-Modeler [9] is a semi-automatic topology generator, based on 3D point clouds. This is an interesting concept, since this method is not about scanning buildings with lasers or randomly generating buildings. The input is in the form of points in 3D space, commonly referred to as a point cloud. CC-Modeler uses these points to fit polyhedron surfaces to, which can result in a wide variety of structures formed. The points themselves can be generated from a human operator, who for example analyzes an aerial photograph of a city, measures buildings and places points, which CC-Modeler then uses to create structures based on these points. This is why CC-Modeler is semi-automatic; the points are generated manually, and the rest is done automatically by CC-Modeler.

3.4 Difficulties/limitations

The techniques mentioned in section 1.1 and 3 are very effective but they have their drawbacks. The first-mentioned technique [1] where data is collected through

laser scanners requires a lot of hardware equipment and vehicles for transport. On top of that, a lot of time has to be spent on research & development since this technique is not on a mass production stage yet, and the devices must be maintained and calibrated. Also, this technique is used to scan already existing buildings, which may not be applicable in games/animations at all, since many games/animations are set in the past or the future, or in an entirely different world, so unless the aim for the game/animation is to replicate an already existing city, it is not going to be useful.

The second-mentioned technique [4] is also good and fast but it shares a distinct drawback with the third-mentioned technique[5]; detail. These techniques, along with Richard Sun's [3], use 3D primitives to construct all the geometry. Differently sized cubes, cylinders, spheres and so on, are used to make facades, supporting columns, windows, rooftops and so on. By no means is this a bad thing, since these techniques are aimed at generating a very large city and this is one of the best ways to do it if one considers the amount of polygons in the scene. If every building in a large city has a couple of thousands of polygons instead of just a hundred or so, it will most definitely be noticed in terms of performance.

Lastly, CC-Modeler [9] requires the input to be in the form of a point cloud. Average computer users might not want to sit down and spend their time on plotting out points for their project. This is also assuming that the users possess the knowledge of how to do so in the first place.

In the next section, a simpler and easier method will be explained. It is not so important for this method to be any better than the methods mentioned above in any aspect, but it is important that this method is easy to understand and provides quick results. The reason for this is due to time constraints. Any further development aimed at improving this method to the point where it rises above the others will continue in future development.

4 Realization

4.1 Preparation

The first week had to be spent with lots of reading and planning, but also on refreshing programming skills/logic. All my previous school projects from the programming classes were studied, in order to remember what I had learned. It didn't take very long before the routine kicked in, and by the second week, experimentations with scripting had already begun.

4.2 Creating the modules

One of the first things that needed to be done was the modeling of the modules. A couple days were spent on working with the 72 modules that the script uses. This number was not a pre-determined goal to reach, but it rather ended up on 72 because it seemed as if it would provide enough variation. These modules are made up of walls, special convex "rounded" corners, concave "rounded" corners, floor, rooftop, ventilation shafts, antenna and other various objects. The most time consuming part was to actually come up with the ideas for the modules and to decide on what style to apply to them. The decision settled on a modern downtown city feel with sturdy yet neat facades with big wide windows. The basis for this decision was that the difficulty of this style was just right to be completed within the given time frame. It is also quite a generic and neutral style, which should be able to find many uses.

The modeling was done in Maya 2009, starting from 2x1 planes consisting of only 4 vertices, as shown in figure 1.

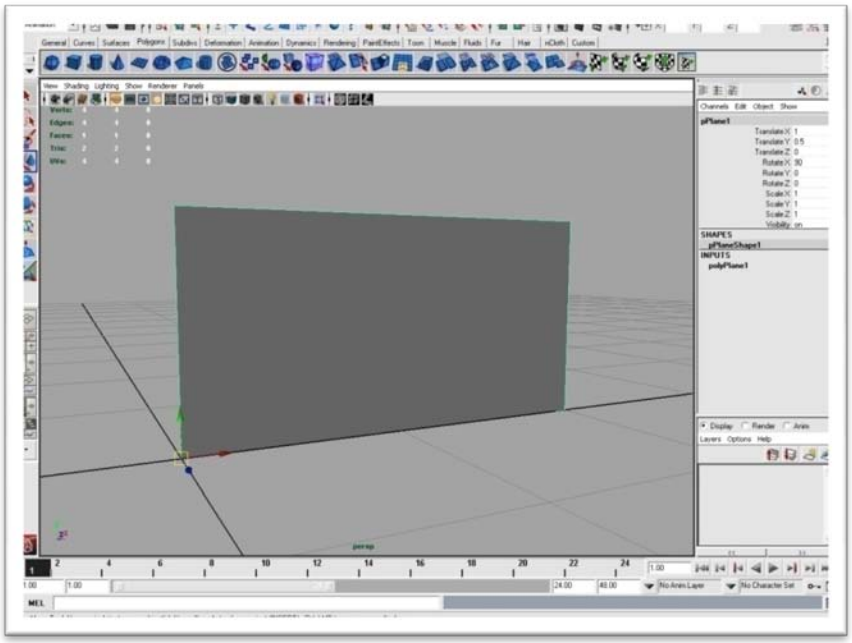

Figure 1: 2x1 polygon plane in Maya 2009.

The most used tools to create the modules were:

Extrude, which is used to extrude the components of the object into new additional components. Polygons (also known as Faces) are the most common component to extrude, and it is a very fast and effective way of adding detail to an object without adding any splits and such.

Polygon splitting tools which are used for splitting a polygon into several, in order to gain more detail to work with.

Vertex snapping, which is used to snap one vertex to the position of another, in order to quickly and precisely model some of the detail, and also to position objects with accuracy.

Merge, which is used to merge two or more components into one. Merging vertices is necessary when two vertices are on top of each other in the same object, otherwise it can cause unwanted effects later.

After a few uses of these mentioned tools, the very first module was created which is a quite basic module with 2 windows, as seen in figure 2.

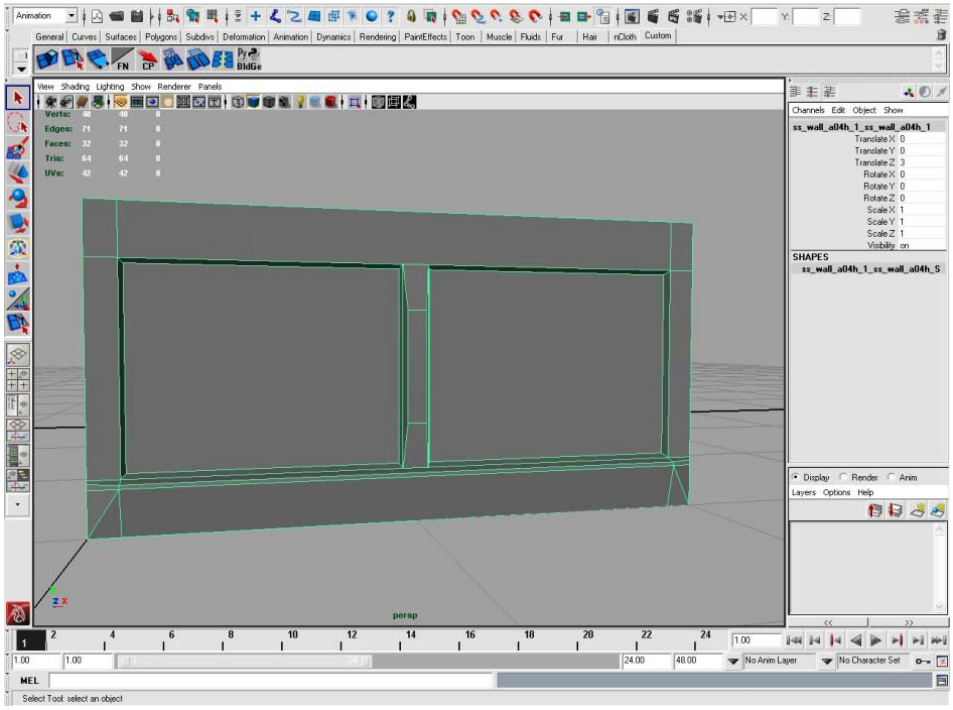

Figure2: The first module.

This work continued until all of the 72 modules were created. Figure 3 shows what the scene file containing all of the modules looks like:

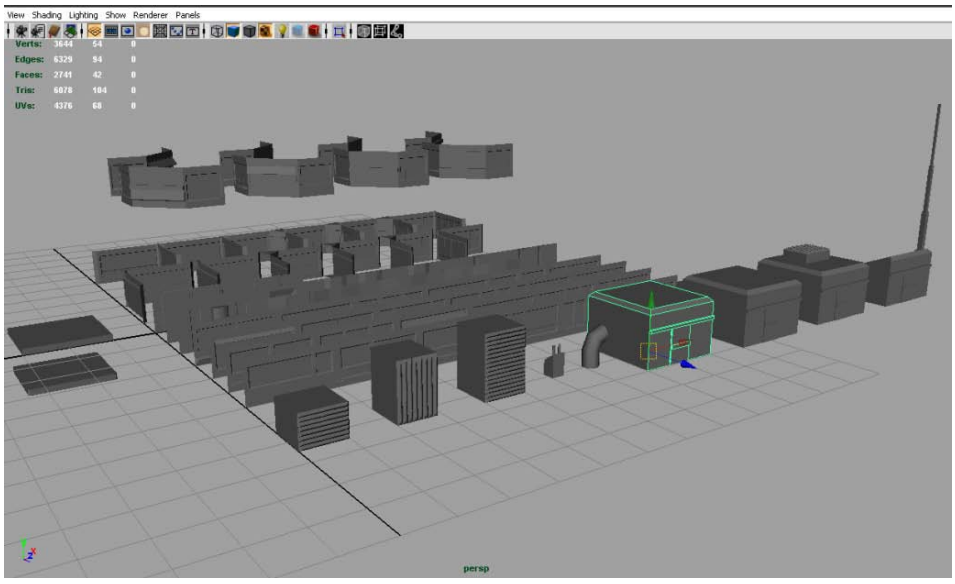

Figure 3: 72 modules. Ventilation shafts and other rooftop objects can be seen in the front, wall modules can be seen behind them and corner modules at the back.

4.3 Importing the script

A problem occurred while trying to load the Python script into Maya. The path to the actual script, the Python file, could not be found. A quick search on Google revealed that one has to put the following line into Maya.env which can be found in "\\Documents and Settings\Your Username\":

```
PYTHONPATH = D:\Erkan\3D\ExProject\mel;
```

This is the line that had to be put into the specific maya.env file for this project. It lets Maya know where to find the python files.

As soon as it was possible to call the Python script from within Maya, the work began on the user interface of the script which contains all the controls. A piece from an old assignment was re-used in order to provide a good solid point to start from, on the interface. Here is a shortened down version of the interface:

```
def createForm():
  global bld_width
  global bld_depth
  global bld_height
  global slg_bld_Width
  global slg_bld_Depth
  global slg_bld_Height

  optionsWindow = mc.window(widthHeight=[400, 600],
    title="TEMP NAME", rtf=True)
  optionsForm = mc.formLayout()

  lblScriptName = mc.text(label="ALSO A TEMP NAME")
  slg_bld_Width = mc.intSliderGrp(label='Building Width',
    field=True, minValue=1, maxValue=20, fieldMinValue=1,
    fieldMaxValue=100, value=5, dc=setBuildingWidth)

  slg_bld_Depth = mc.intSliderGrp(label='Building Depth',
    field=True, minValue=1 , maxValue=20,  fieldMinValue=1,
    fieldMaxValue=100, value=5, dc=setBuildingDepth)

   slg_bld_Height = mc.intSliderGrp(label='Building Height',
    field=True, minValue=1, maxValue=50, fieldMinValue=1,
    fieldMaxValue=1000, value=5, dc=setBuildingHeight)

  btnCreateBuilding = mc.button(label="Create Building",
    command=createBuilding)

  mc.formLayout(optionsForm, edit=True, attachForm =
    [
    (lblScriptName, "top", 15),
    (lblScriptName, "left", 150),
    (lblCornerInfo, "top", 200),
    (lblCornerInfo, "left", 190),
    (btnCreateBuilding, "top", 560),
    (btnCreateBuilding, "left", 15),
    (slg_bld_Width, "top", 50),
    (slg_bld_Width, "left", -35),
    (slg_bld_Depth, "top", 75),
    (slg_bld_Depth, "left", -35),
```

```
        (slg_bld_Height, "top", 100),
        (slg_bld_Height, "left", -35)
        ])

    mc.showWindow(optionsWindow)
```

This script defines a function which is called from within Maya, in order to bring up the interface of the script.

Global variables are set in order to tell Python that these variables should not be local to this function only, but rather be available for all of the script. This is necessary in order to set variables that will be used outside of any given function.

The interface window is created with the mc.window function and the mc.formLayout function is used to arrange the layout of all the elements in the window that was created.

Sliders were created by using the mc.intSliderGrp function which only returns integers when dragged. There are sliders that return float numbers as well but in this case, integers make the most sense to use.

Buttons are created with the mc.button function. This function has a parameter called “command”, which links the button to a function in order to make it perform a task. In this case, the “Create Building” button which is the only button in the interface is linked to the “createBuilding” function, which is the core of the script. That function is responsible for importing and arranging the modules into buildings.

Although no checkboxes are present in this excerpt of the script, they are created with the mc.checkBox function which also has a parameter to link it to a function. More on this will come in section 4.4.

4.4 How the script works

The script's inputs are taken from the sliders and checkboxes that make up the user interface of the script. The user simply drags the sliders to increase or decrease the values that control the width, depth, and height of the building, and the values to control the various rooftop objects such as ventilation shafts, rooftop doors and antennas. The width, depth and height are measured in modules, so a width of 5 would mean 5 modules wide, and since each module is 2 units wide, it would mean 10 units. Figure 4 shows the user interface.

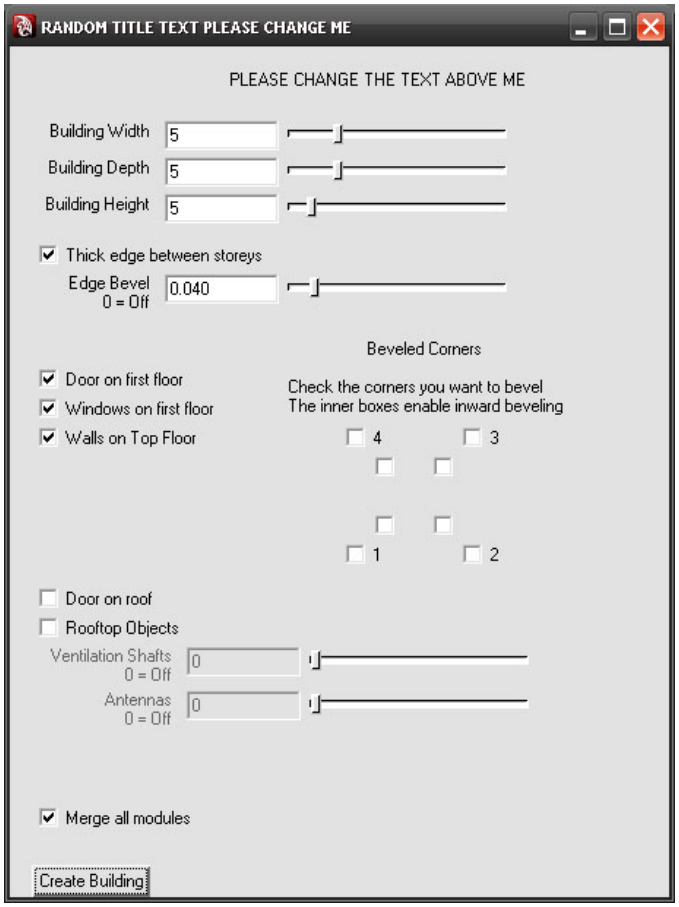

Figure 4: A non-final version of the UI.
The naming labels need an update to become a bit more "elegant".

The checkboxes in the middle, slightly to the right, control the appearance of the corners. The design of this is only temporary of course, but as of right now, it works by clicking the boxes to activate which corners that will get beveled or rounded off. The boxes represent the building seen from top, and the outer boxes enable the normal beveling, while the inner boxes make the bevel or "rounding off" go inward instead. This feature has no effect unless the outer box is checked, since it also acts as an on or off switch for the whole bevel effect.

Technically, when checking the boxes, they send their value to a function which checks the state of the checkbox, and then set a variable from "0" to "1" which corresponds to its corner. If-statements are used to check which corner has this variable set to "1", and the script then imports one of the appropriate corner modules instead of using the standard wall modules.

Here is an example of the code that checks the value of corner checkbox 1:

```
def setCorner1(value):
    global corner1
    if value == "true":
      corner1 = 1

    if value == "false":
      corner1 = 0
```

The variable “corner1” is checked later in the script to find out whether or not a corner module is to be imported. Here is an example of that code

```
elif i == 0 and corner1 == 1 and invCorner1 == 0:
  mc.file("ss_corner_wall_start_a04h_%d.mb" %rndModCorner,
  i=True, dns=True, type="mayaBinary")

elif i == 0 and corner1 == 1 and invCorner1 == 1:
    mc.file("ss_corner_rev_start_wall_a04h_%d.mb"
  %rndModinvCorner, i=True, dns=True, type="mayaBinary")

else:
  mc.file("ss_wall_a04h_%d.mb" %rndModWall, i=True,
  dns=True, type="mayaBinary")
```

This is an excerpt of a larger if-statement, which checks if “corner1” equals to “1”, in which case it imports a corner module. If “corner1” and “invCorner1” is “1”, a reversed corner module is imported. In any other case, it imports a straight wall instead.

The core of the script, as mentioned, is the “createBuilding” function. This function takes the values from the user’s inputs, and uses them to create the corresponding building. It consists of mostly for-loops and if-statements, with an occasional while-statement for things like simple collision detection. The numbers that are returned from the sliders that set the dimension of the building are used to determine how many modules the script should import. If width is set on 5 for example, it simply means that the building will be 5 modules wide. Here is a shortened down example of one of the most important pieces of the script:

```
for i in range(0, bld_height):

  for i in range(0, bld_width):

    mc.file("ss_wall_a04h_%d.mb" %rndModWall, i=True,
    dns=True, type="mayaBinary")

    mc.select("*a04h*", r=True)
    mc.rename("tmp_mod%d_1a2b3c" %iD)
    mc.move(moveXZInc,moveYInc, 0)
    moveXZInc+=2
```

There are 3 more loops nested in this one, making it a total of 4 nested for-loops. This is because the building has 4 sides, so there are 4 for-loops with one creating the facade that points in X-positive, one in X-negative, one in Z-positive and lastly one in Z-negative. The example shown above has been shortened down; otherwise it would span over a couple of pages. The most important functionality is still there though, and that is to go through the loop from 0 to bld_height, which is the number the user inputs via the slider to control the height. This makes sure to add as many storeys as the user input. The same thing goes for bld_width. After that, the module is imported, selected

and renamed in order to avoid being selected when the next module is imported. Then it is moved by variables moveXZInc and moveYInc that increase every time a loop is completed so that there is an increment. Otherwise, the modules would never progress, and they would end up being stacked on top of each other.

The simple collision detection is only used for the ventilation shafts on the rooftop. It checks every ventilation shaft and compares it to every other ventilation shaft on the same roof, and if one of them is within 1 unit in either X or Z of another, it will randomly re-position them again, until they are spaced out from each other. Here is an example of that code:

```
if rooftopObjects == 1:

  if ventilationShafts != 0:

    for iDx in range(1, ventilationShafts+1):

      rndRoof = rand.randrange(1, 6)

      mc.file("ss_rooftop_object_a04h_%d.mb" %rndRoof,
      i=True, dns=True, type="mayaBinary")

      mc.select("*a04h*")

      mc.rename("tmp_rftdox53a_vent_shafts1")
      mc.move(round(rand.uniform(2, bld_width*2-2), 2),
        bld_height+0.15, round(rand.uniform(-bld_depth*2+2,
        -2), 2), ws=True)

      for i1 in range(1, iDx+1):
        checkX = mc.getAttr("vent_shafts%d.translateX" %i1)
        checkZ = mc.getAttr("vent_shafts%d.translateZ" %i1)

        if i1 != iDx:

          while
            mc.getAttr("vent_shafts%d.translateX" %iDx) >
            checkX-1 and
            mc.getAttr("vent_shafts%d.translateX" %iDx) <
            checkX+1 and
            mc.getAttr("vent_shafts%d.translateZ" %iDx) >
            checkZ-1 and
            mc.getAttr("vent_shafts%d.translateZ" %iDx) <
            checkZ+1:

            mc.move(round(rand.uniform(2, bld_width*2-2), 2),
              bld_height+0.15, round(rand.uniform(-
              bld_depth*2+2, -2), 2), ws=True)

      mc.rotate(0, (90*rand.randrange(0,4))+rand.uniform(-
        2.5, 2.5), 0)
```

Even though it is very hard to format code to fit into such a narrow document like this one, we can see the nested if-statements and for-loops, and the while-loop. The while-loop is the one that determines if the objects are too close to each other, in which case it randomly moves them around until they are not too close anymore.

5 Results

Here are a few examples of buildings created with the script by only adjusting building dimensions (width, depth and height), corner checkboxes and the “Windows on first floor”-checkbox. The render comes from Maya using the Mental Ray renderer. There is nothing special to mention about the rendering, it is the most basic setup using only the “Physical Sun and Sky” lighting model in Mental Ray.

The first building is made with the default settings in the script (figure 4 shows the default settings), and an image of this building can be seen in Figure 5.

Figure 5: Building made with the default settings.

The next building, shown in Figure 6, was made by altering the settings slightly. The width and depth of the building is set to 4 while height remains on 5. The checkboxes to control the corner rounding/beveling were checked, but the inner checkboxes were not checked. This means the corners are going to be rounded in a convex manner.

Figure 6: Width and depth on 4, height on 5. Convex corner rounding.

The next building has width and depth set to 3, while height is set to 12. The inner checkboxes have also been checked for all corners, which will now cause the “rounding” to be concave instead, seen in figure 7.

Figure 7: Width and depth set to 3. Height set to 12. Concave corner “rounding”.

The next building was made with the same width and depth as the last one but with height set on 16 instead. Additionally, the “Edge Bevel” parameter linked to the checkbox “Thick edge between storeys” was maxed out to 0.400. This is rounding off the corners of the floor that is extending slightly outside the walls, which is creating the defined “storey” look. Also, 2 antennas were added via the “Rooftop objects” checkbox, and the “Antennas” slider. The checkbox “Windows on first floor” has been checked off, which creates a concrete foundation for the first floor instead. This can be useful for big buildings which appear as if they need a heavy foundation, as the building seen in figure 8 for example.

Figure 8: Width and depth set to 3, height set to 16. Concave corner “rounding”, “Edge Bevel” set to 0.400, “Windows on first floor” checked off, “Antennas” set to 2.

Here is an example of a building created with the same settings as previously in Figure 8, except for the “Thick edge between storeys” checkbox being checked off. This means that there are no floors that are dividing each storey now. The result can be seen in figure 9.

Figure 9: Width and depth set on 3, height set on 16. “Thick edge between storeys” checkbox is checked off.

With bigger values on width and depth, it is possible to produce buildings like hospitals and such. The next example has width set to 5, depth set to 12 and height set to 8. The corners are a bit different from each other; there are both concave and convex corners. There are also 10 ventilation shafts, 1 rooftop door and 2 antennas in the settings. Figure 10 shows this building.

Figure 10: Width 5, depth 12, height 8, mixed corner types, 10 ventilation shafts, 1 rooftop door and 2 antennas.

It is also possible to create other buildings and placing them right next to another building in order to create more interesting and intricate building shapes. The following example uses the exact same “hospital” building as in Figure 10, but with 3 additionally added buildings. This picture basically wraps up Figure 5 to 10 in picture. It contains a skyscraper-like building, a big main building and a couple smaller buildings, all attached to the main building. By using grid snapping in Maya, it is extremely easy to place the buildings tightly against each other, as shown in figure 11 and figure 12.

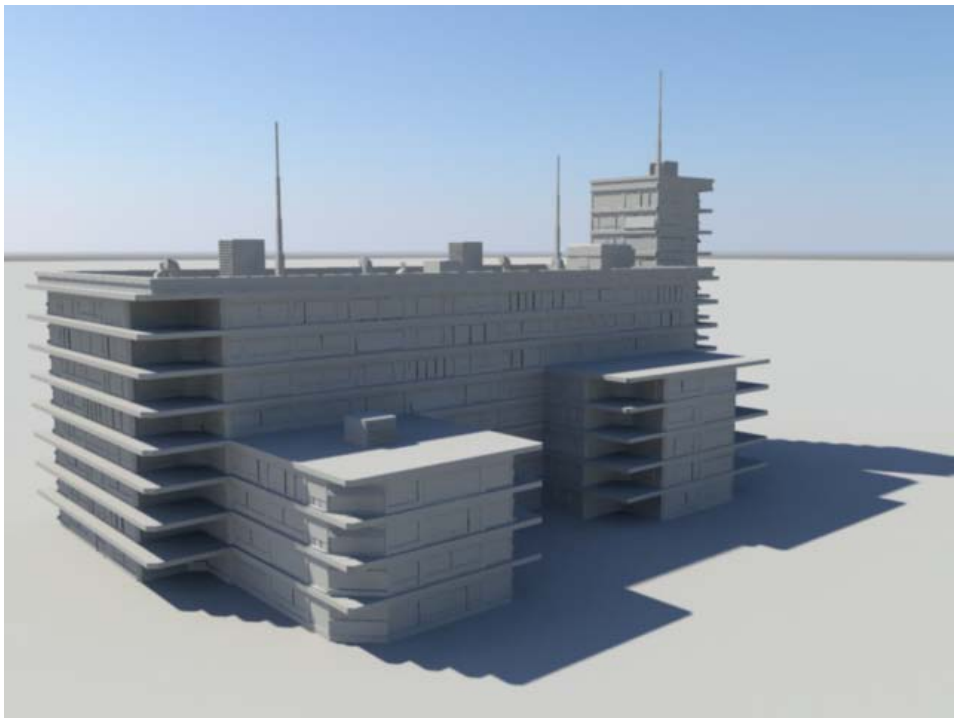

Figure 11: Several buildings attached to one. This could represent a hospital.

Figure 12: Same building, different angle.

Figure 13 to figure 15 show one building each, but the buildings all have the exact same dimensions, which make them look like they are the same building. The point of this is to illustrate what happens when a building is created several times with the exact same settings. It practically is the same building, but none of them look identical because the modules are constantly shuffled. This means that no matter if you create buildings of the exact same shape several times, they will not look exactly the same. If the script did not randomly choose among all the modules when importing, this building would look exactly the same on figure 13-15.

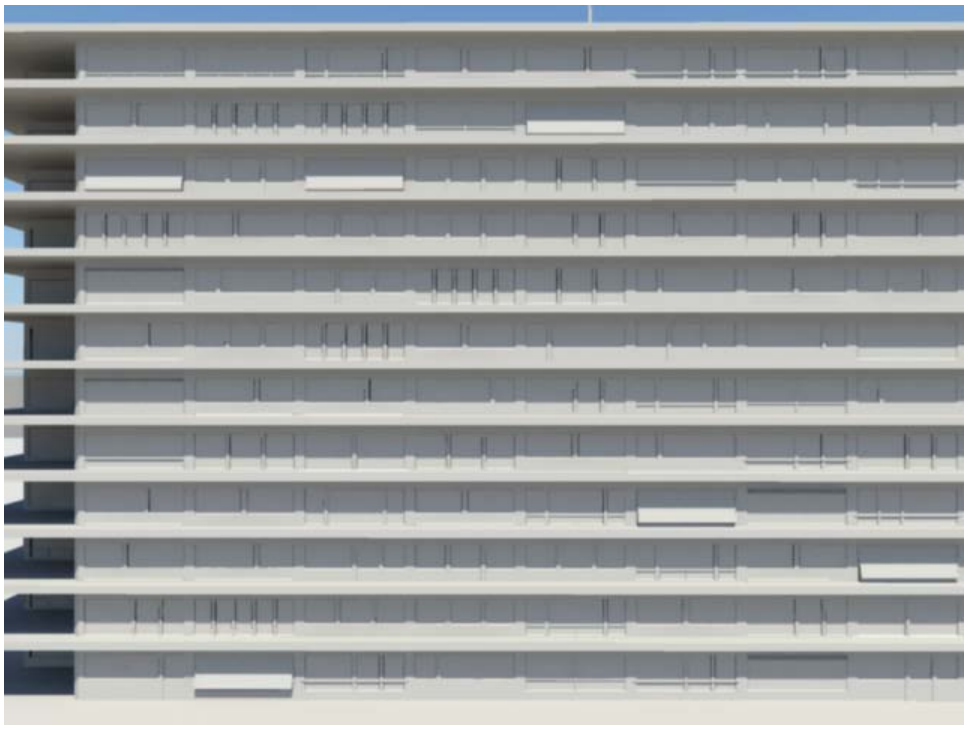

Figure 13: A large building, seen from the side.

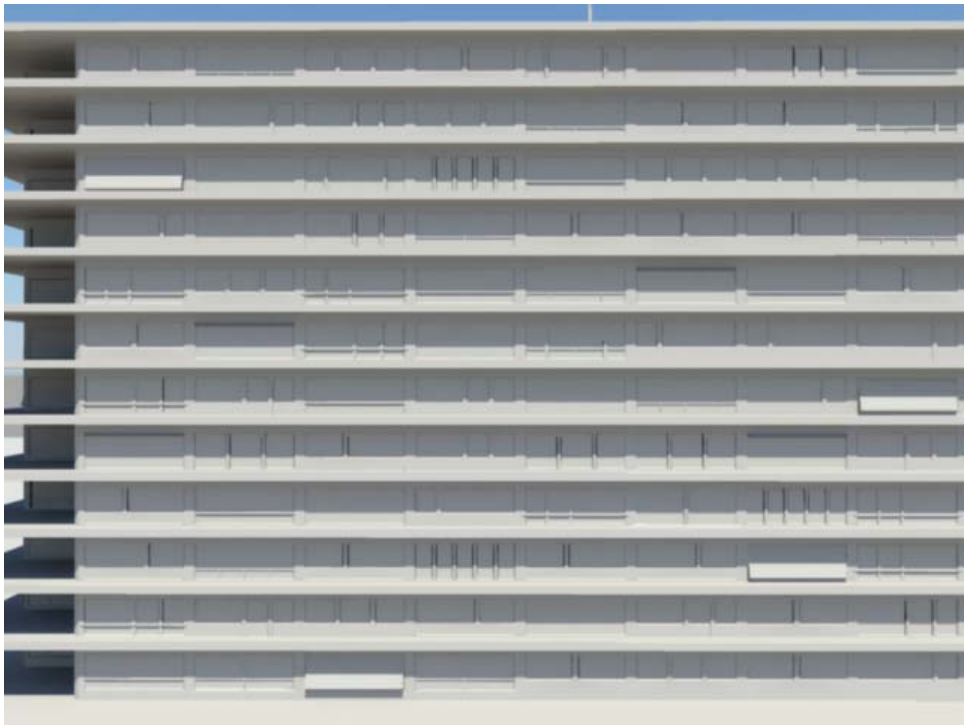

Figure 14: Another building with the exact same settings as Figure 13. The modules are not in the same order as in the other two images.

Figure 15: Yet another building with the exact same settings as Figure 13. The modules are not in the same order as in the other two images.

All the figures so far in this section have been showing single buildings. We can see that there are a range of different shapes which can be created, and even more if one combines different buildings with each other, as in figure 12, to create something a bit more complex, like a hospital. But now it is time to test how big of a city it is possible to create with this script in its current stage.

There is a movie which demonstrates the script in action with a rendered animation, and it can be found at the reference page [8]. However, the movie only shows a small “district”, but figure 16 to figure 18 show much larger cityscapes.

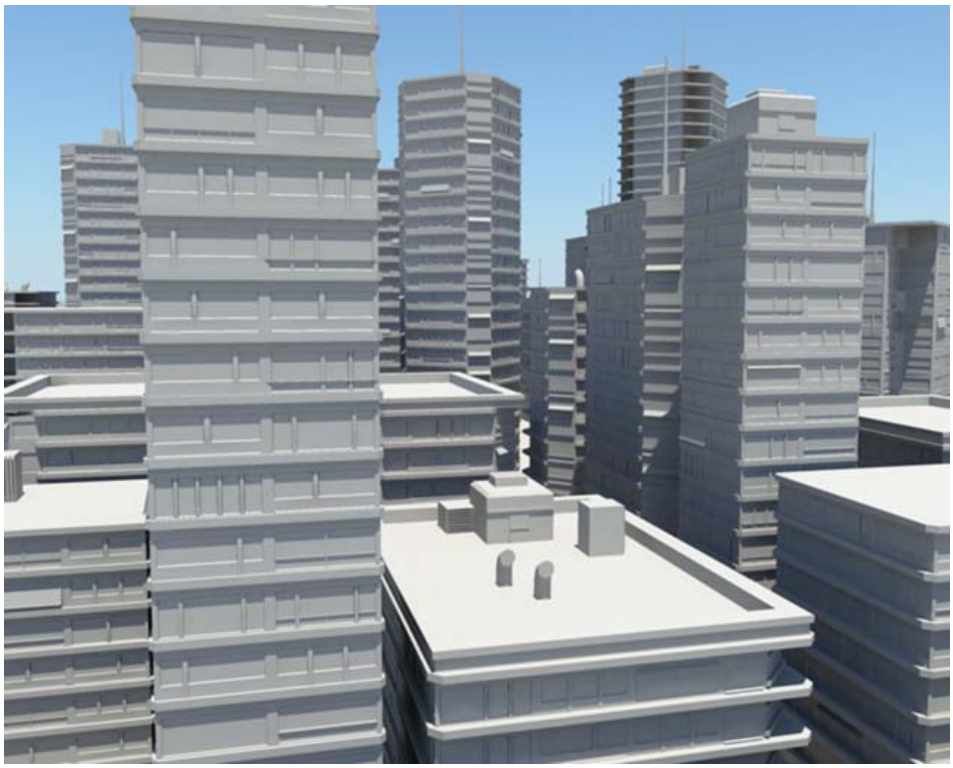

Figure 16: A fairly dense city core with large buildings.

Figure 17: Looking down the main road to the city core.

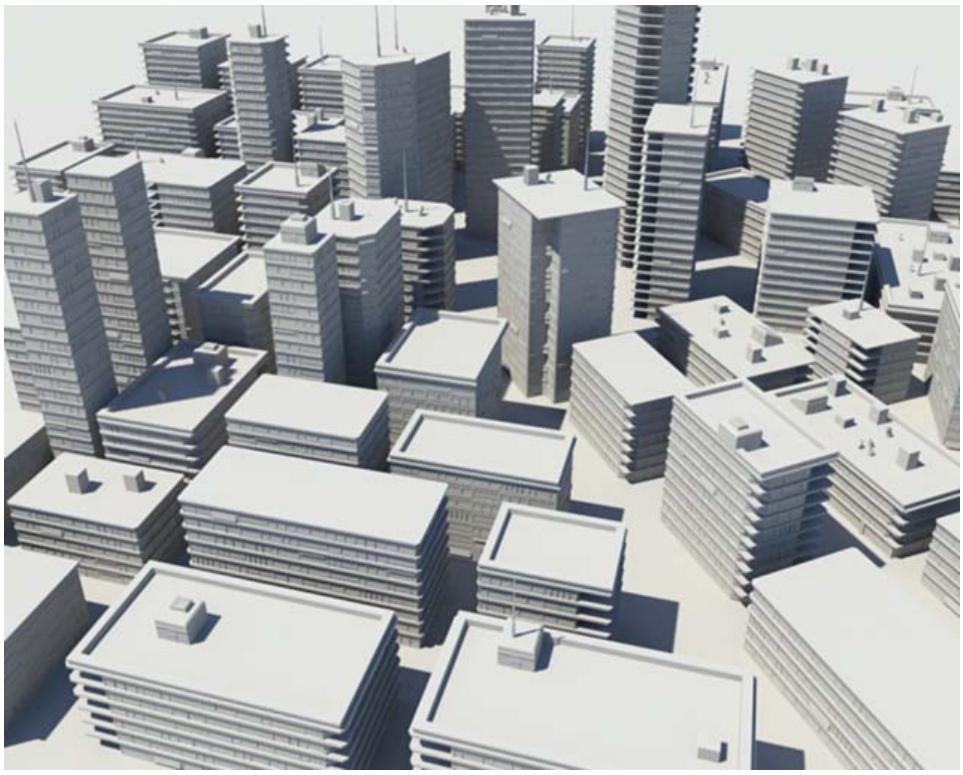

Figure 18: A bird's view on the city.

Figure 16 to 18 show a city size which is very manageable on an average computer of today's standards. This means a computer with an Intel Core 2 Duo 2.13 GHz processor, 2 GB RAM and an ATI x1950 Pro graphics card. Figure 19 to 20 show a city which is beginning to push the limits of this computer.

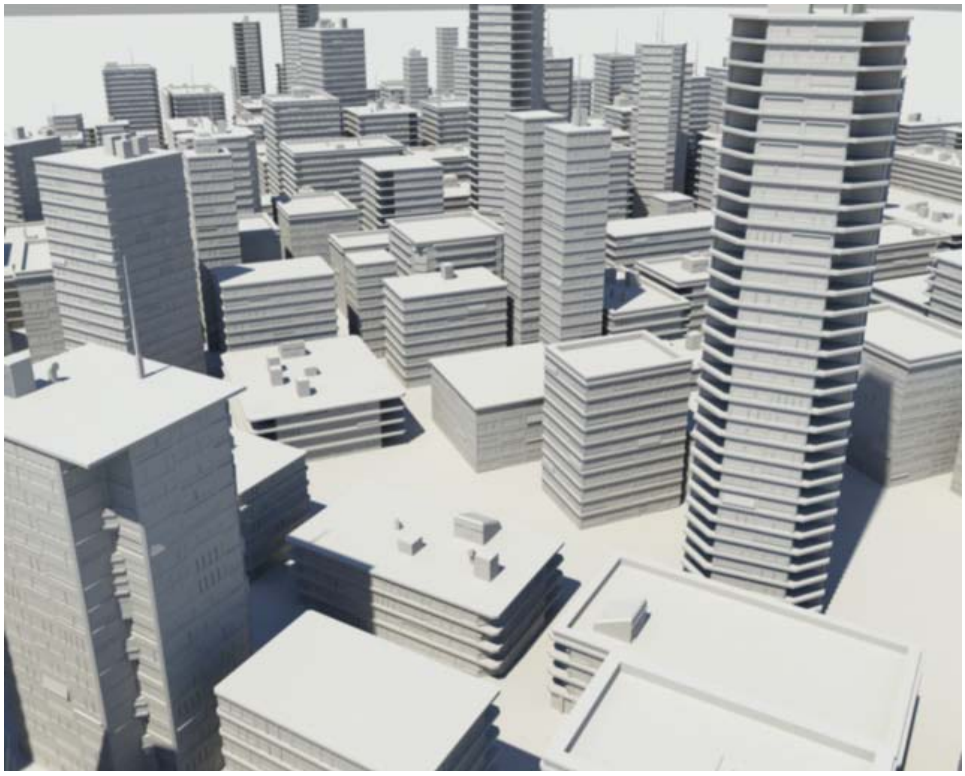

Figure 19: More buildings have been added towards the horizon.

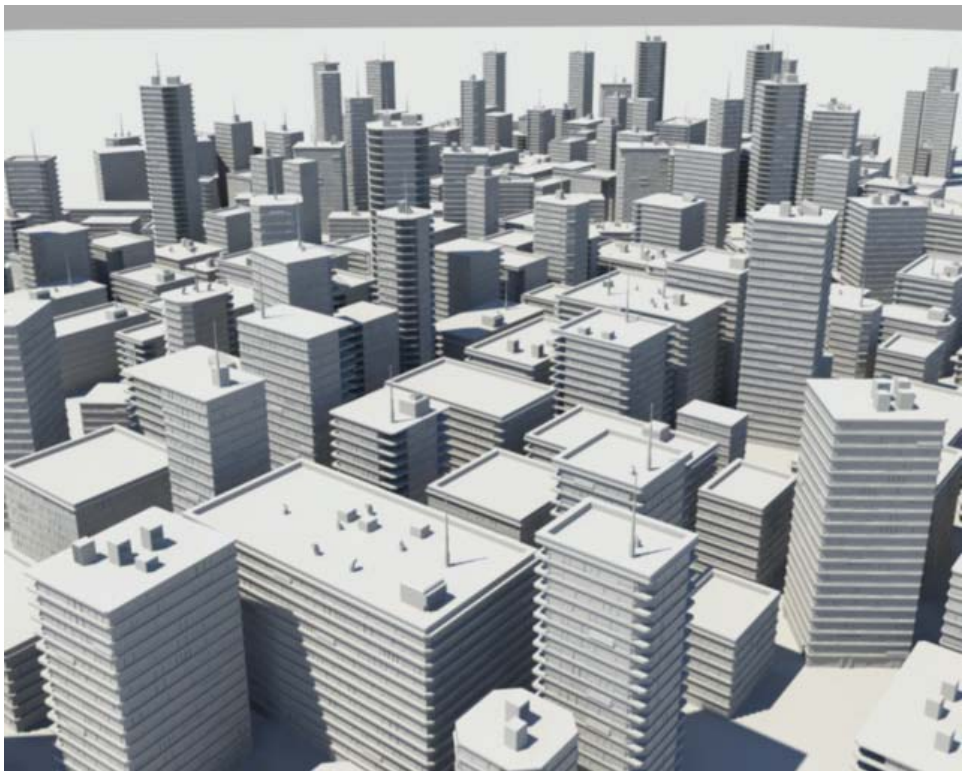

Figure 20: Even more buildings added towards the horizon.

On figure 20, there are about 1.5 million polygons which do push this computer's limits. Even though the script is not designed for mass generation at the moment, it does work fairly well regarding big city generation. Some duplication was used here however, because creating and placing the buildings one by one would take a lot of time. Figure 16 to 18 were created without duplicating anything, but figure 19 to 21 contain duplicate buildings. However, it is rather hard to spot them on these pictures without textures and with low resolution.

Figure 21 contains about 3 million polygons, and it is closing in on the limit for this computer. It is possible to go higher and create an even larger city, but then the troubles with rendering begin. Even if it is possible to have that many buildings in Maya, it is not certain it will be renderable, especially in Mental Ray, due to exceeding the memory limit. Figure 21 was not renderable in Mental Ray, and was therefore rendered with Maya Software Renderer.

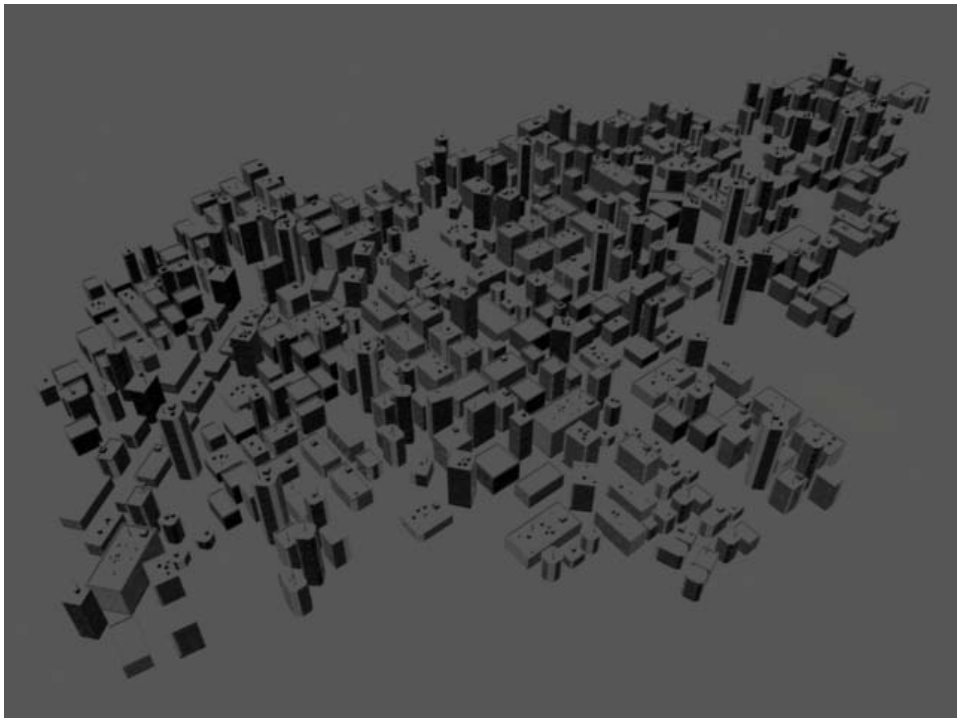

Figure 21: Big city, starting to breach the computer's limits.

Figure 21 contains over 3 million polygons and does not render in Mental Ray due to too low RAM in this computer. This render is a quick Maya Software render with default lighting. The city is about 1.3 kilometers long and 0.7 km wide. It took about 45 minutes to do the layout of this city.

6 Discussion

It seems like this module based generator does have potential. It is possible to build quite large cities, but since the script is not designed for that at this moment, it does take a while to do so. This is because the script only generates one building at a time, but it is very likely that a further development will give birth to a system which creates multiple buildings at once, which should bring this script a little bit closer to the other ones [3] [4] [5] in terms of mass production.

The advantage with this module based method is that nothing is left for the computer to decide when it comes to style, which means artistic freedom can be retained even in such an automated process. By simply going in and editing the modules, one can decide how it is going to look down to the smallest details. It is also easier to work with, since what you see (the modules) is what you get, as opposed to a system where the computer generates everything from scratch. A lot of time would be spent on steering the computer in the right path, telling it how not to do, rather than how to do.

The disadvantage is that the system is not very flexible and there are not many ways to affect the looks once the modules have been created and saved. Even though it

is possible to enter the component mode on a module after it has been imported in order to change something about it, it will still not be very useful since there are no globalized parameters like “window size” for example. If the script was not based on modules, it would have parameters to control how many windows, how large windows, what type of walls and so on, which would be a quick way to alter the looks. But then again, it would be all up to the computer so there would not be much room left for artistic freedom in the design.

6.1 Encountered difficulties

Although shading/texturing was not in the initial goal for this time frame, there were a couple of days spent on trying to apply it on the buildings to further improve the appearance before the presentation week. The method was to create all the shader nodes needed within the main Maya scene file where all of the 72 modules are stored. From there, the shaders were individually applied to all of the modules. For instance, every window module consists of windows, the metallic windowpane and the surrounding facade, which requires three different materials to be applied to their respective polygons. Once done and saved, each module was exported back out into its own file which was then used to import as usual by the script. The script went about and created a building as usual, but some of the vertices of some of the modules kept moving with the looping construction procedure, which resulted in something which looked like a whirlwind of vertices, continuing around the building.

The solution to this problem was found at the same time as writing this section in this report. The problem was the small “tag” that was added in each module’s name, which was used for selection in Maya. Changing that “tag” from “a04h” to something else solved the issue. Due to time constraints, the shading could not be applied anyway but as said earlier, it is on the to-do-list instead.

However, there is more to this problem than mentioned. Each module gets its shaders applied to it in the main Maya file, as mentioned, and when exported, each module also gets a copy of their nodes which is stored in every module file. When the script later imports the modules, each module brings its copy of its shaders, which results in as many duplicate shaders as there are modules in the scene. The option “Remove duplicate shader networks” in Maya’s import options does nothing to remove these duplicate nodes, which is a big setback. Even though the shaders do not necessarily have to be in the module files, the information of which polygon gets which shader needs to, since it is very difficult to assign the shader after the importing since there is no way to know which polygon is a window or which polygon is a windowpane/facade and so on.

However, I discovered a way to solve this. To assign a specific shader, for example the window shader, to a specific polygon of a module, it is possible to create Selection Sets within Maya. This solution is a good way to solve this problem where the shaders do not need to be in the module files, but the information of which face gets what shader needs to. Selection Sets are basically nodes that contain information, which also get exported with the object when exporting. It is for example possible to select any number of polygons of an object and then making a Selection Set of it. This will store that information, and by selecting the set, all of the stored polygons are selected. Unfortunately, after making selection sets for half of the modules, I noticed Maya had obliterated all the work done so far. When going back to check if the Selection Sets contained the correct information, they had mixed up all the polygons among them. The Selection Sets containing all windows for example, contained various polygons from the rest of the object, in a very erratic and random pattern. Why this occurs, is still a mystery.

6.2 Personal reflections

Personally, I am happy with the outcome of this research, even if there are many improvements and refinements left to make. I have learned a lot of things, especially when it comes to Python. It has been fun, and it has been very challenging, but when I get to see the images of the buildings that my script has generated, it motivates me to keep coming up with new ideas and to keep improving the script.

I do believe that this script can become very useful in the future when the additional improvements (listed in the section below) are implemented. I have not seen any close-up renderings of the referenced articles/techniques, which indicates that I might have an edge here. Of course, this still requires a lot of work.

The results are somewhat what I expected, which of course is a good thing. I was not worried about the results at all while working with the script since I could see for each line of code that I was getting closer to the goal. I have been wondering how long it would take if I made Figure 21 without this script, and I fear it would be many hours so it might be best not to know.

7 Future development

There is always room for improvement, and the list can get very long. Here are the most eligible candidates for future upgrades:

7.1 Texturing

A method for texturing the buildings would definitely be one of the first things to get developed. What is needed here is a system to quickly identify certain polygons of the modules, for example all polygons that are windows would be selectable in a quick and easy manner. A way to achieve this is to create Selection Sets, but for some reason this is really unstable in Maya 2009. As discovered and mentioned in section 6.1, the Selection Sets function in Maya ”forgets” and mixes up polygons within these Selection Sets as the user continues to create sets and assign polygons to them. This is not a random occurrence; it was possible to reproduce this faulty behavior several times. In any case, if this issue could be solved, it would be easy to assign the appropriate shader for the different Selection Sets in the modules.

This texturing method would be further developed to reflect the non-identical feature of the script, meaning the way the script works in order to construct buildings that are never exactly the same. It would be boring if all buildings had exactly the same shaders, so in order to solve that, a method to slightly change texture values would be developed. This could be as simple as randomly changing the color/specular/bump map of each shader, or as advanced as letting the users choose which material they want on the buildings, and controlling the exact values of them.

7.2 Creating a new construction algorithm

At the moment, the way the script constructs the buildings from the user’s inputs is fairly simple and straight forward. The user lets the script know the dimensions and any extra features, and the script generates the building. However, control is the keyword and therefore it is very likely that the core algorithm of the script will be remade. A couple of ideas spring to mind, one being a way for the user to “talk” to the script via a “language”, in order to exactly determine the shapes of the buildings to be created. This could be as simple as the user typing in letters in a text field, where each letter calls a specific module. Then, by typing sequences of letters, the user can make custom shapes. This is a bit clumsy and not so easy to understand perhaps, so a better but more advanced way of doing it could be by using images.

By using a bitmap, the user could essentially “paint” the shape of the building by simply drawing the shape of the building seen from the top view. The script would then analyze this image and calculate how many modules are required in the X and Y-axes. The height could be set via a simple slider, or the same technique could be applied again, to draw the building’s shape from the side view, essentially covering all three axes.

7.3 Multi building generation

At the moment, it is quite tedious to build anything on a larger scale than a few streets/blocks, since the buildings are generated one by one by the user’s inputs. An improvement could be a function that allows the users to edit a few more parameters which randomizes the basic parameters of the script, and creates as many buildings as specified by the user. This amount of randomness would also be controllable so that the users can choose to create for example 10 skyscrapers of the exact same dimensions, or 10 skyscrapers which are allowed to vary wildly in their dimensions and any extra features.

7.4 Module expansion packs

Future expansion packs is naturally something that would be developed for this script. The current script uses 72 modules that have a modern city style, but the possibilities are many. It is actually quite easy to model another 72 or even more modules, as long as one comes up with the design idea. Once this is done, it would be possible to create for example older looking buildings like brick buildings or even older ones that are made of wood and so on.

7.5 Dynamic simulation

Another interesting possibility is to take advantage of the fact that the buildings are already made of small pieces - modules - which could be used as a basis for any dynamic simulation such as fragmenting and exploding the buildings. This is possible to do even now at this stage, by simply clicking off the “Merge modules” checkbox, but the buildings’ smallest components will still be the modules, which will probably not look good in a dynamic simulation. In order to solve this, a method to fragment the modules into even smaller pieces would need to be developed. This should not be too hard though, since it can be done by using a polygon plane with a noise on its vertices to make the plane’s surface uneven and jagged. Then, by using Boolean operations within Maya, it is possible to split the object up from various angles by rotating the plane. This technique is used by Richard Sun in his MEL Cityscape Generator [3].

7.6 Additional detail generation

Another thing to develop would be a system to create various intricate designs and patterns, like arcs and spiral stairs or cylindrical towers or even indoors decoration with furniture. This is something that would require a lot of research though, because there are so many ways these details could be created, it would be necessary to find the most appropriate one. Maybe some details are better off to simply be created “as is”, for example as a pre-modeled object which is ready to be placed anywhere in the scene, rather than letting the users have control over many inputs for that little object. Or maybe some objects should take input from the users, for example bridges in the cities. Specifying the length, width and height of the bridge, along with how many traffic lights, wires, maintenance sheds and so on could be a good thing. However, giving the users control over too many things could prove to be dangerous and block creativity and generally swamp the users in unnecessary information management.

References

[1] Zakhor, A., & Früh, C. *"The speedy way to capture a city"*, http://www-video.eecs.berkeley.edu/~frueh/3d/ (2009-05-25).

[2] McDonnell, R., Larkin, M., Dobbyn, S., Collins, S., & O'Sullivan, C. *"Clone Attack! Perception of crowd variety"*, International Conference on Computer Graphics and Interactive Techniques ACM SIGGRAPH 2008 papers, Los Angeles, California SESSION: Characters, Article No. 26, Year of Publication: 2008 ISSN:0730-0301 http://portal.acm.org/citation.cfm?id=1360612.1360625 (2009-06-06).

[3] Sun, R. *"MEL Procedural City Generator & Custom Tools"*, http://richsunproductions.info/vsfx705/modelingsuite.html (2009-05-25).

[4] Colefax, C. *"City Generator for POV-Ray"* http://www.geocities.com/ccolefax/citygen.html (2009-05-25)

[5] Introversion, Inc. *"Subversion's procedural city generator"* http://forums.introversion.co.uk/introversion/viewtopic.php?t=1132 (2009-05-25)

[6] Python.org *"Information about Python"* http://wiki.python.org/moin/LanguageComparisons (2009-05-25)

[7] Blender Foundation, *"Blender 2.48"* http://www.blender.org/ (2009-05-25)

[8] Dogantimur, E. *"Building Generator in Python" (2009-05-25)* http://www.youtube.com/watch?v=vkzzaTcAZk4

[9] Gruen, A., Wang X. **"***CC-Modeler: a topology generator for 3-D city models*" ISPRS Journal of Photogrammetry and Remote Sensing Volume 53, Issue 5, October 1998, Pages 286-295