TITLE: User-Designed Facades

Anton Khabbaz & Yuru Wang

PROJECT SUMMARY

Artists need a way to create realistic buildings from a template and one of the best ways to proceed is to start with pictures of typical buildings. Typically, in procedural modeling, a grammar is used to generate a building or a façade. That approach suffers from a shortcoming that it is difficult to understand the implications of grammatical rules, and it is difficult to modify the rules a bit to produce specific types of façades. Here, instead the inverse problem is tackled, where a marked-up façade is used to generate grammatical rules which in turn generates façades. This work implements the procedure in "Inverse Procedural Modeling of Facade Layouts", by Wu, Yan, Dong, Zhang, and Wonka, SIGGRAPH 2014.

The design goal is to take pre-described façades and that have marked terminal regions such as windows and doors, and produce a grammar that closely reproduces the façade. This reproduced façade should appear similar but not identical to the original façade. It should have the specific features or details of real facades. The target audience will be game and animation studios that will use this to create more realistic cities. The tool will have a GUI allows the user to select a model façade, and then the user can select the type of door and window.

The tool will take an marked up input image and generate a small grammar that would produce the facade. That grammar then would produce realistic facades just like the original one.

We first plan to take marked up input facades and produce the grammar as the paper did. Producing the grammar is the heart of the application and that will be done by alpha. Next, by beta we will have the GUI working and the user will be able to generate facades and buildings. By the final version the user will have more control over the selection and editing process and will be able to mark up their own images and have more options about how the facade is built.

1. AUTHORING TOOL DESIGN

1.1. Significance of Problem or Production/Development Need

What is the significance of the problem and/or the production/development need for the functionality or special effect you plan to develop.

There are many procedural modeling tools for buildings out there but few of them could produce natural looking facade. Artists might need to do manual modifications to the generated models to make them looks realistic, and even to generate a set of grammars by themselves. So there is a need for a more efficient way to generate procedural buildings with natural looking facade.

In application scenarios where artists need to build large scale city models or construct urban scene, they need a large number of variations of procedurally generated buildings, and also a bunch of variations for facade which still preserve semantically meaningful layout. Hence we also need capabilities for generating facade variations efficiently.

1.2. Technology

This paper proposed a method for evaluating split grammars with cost functions and they described algorithms that can extract semantically meaningful split grammar from a facade layout input, which is exactly what we need to produce procedural building models with natural looking facade.

This authoring tool will implement the algorithm described in the papar, and construct a system which takes a 2D facade layout input and extract meaningful split grammars from it, and finally procedurally generate building models using those grammar. So the output of the system would be building models with facade that looks similar to the input facade layout

1.3.1 Target Audience.

Target users for this authoring tool could be artists and designers in animation studios and game studios.

1.3.2 User goals and objectives

The users are able to create building models with very natural looking facade as the facade layout they provide. They are also able to produce a bunch of variations from the existing facade layout and create a number of different building models which still looks realistic. This could be extremely helpful when the user need to build large scale procedural city and urban scene reconstruction.

1.3.3 Tool features and functionality

- Extract semantically meaningful split grammar from input facade layout
- Produce natural looking building models with generated grammars
- Generating variations of facade
- Customize buildings with user specified elements models

1.3.4 Tool input and output

Input:

An 2D facade layout as input

`

Output:

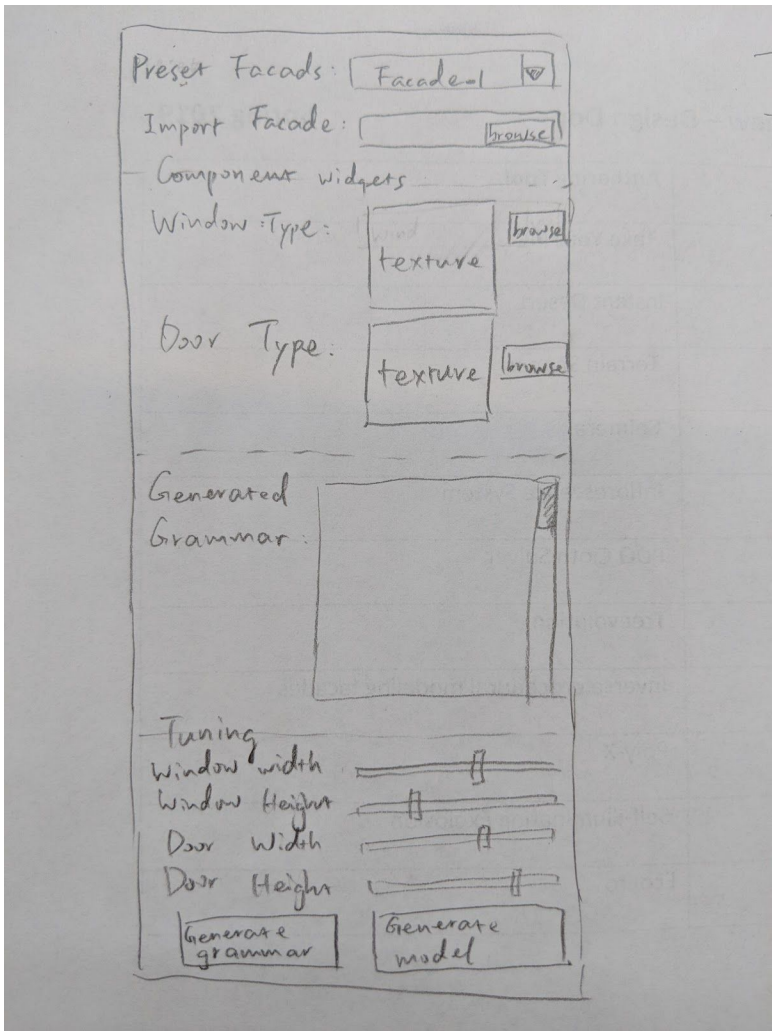
A set of split grammars extracted from the input layout
Building models procedurally generated using the grammars
A bunch of buildings models with facade variations

1.4. User Interface

1.4.1 GUI Components and Layout

There would be dropdown menu item for this authoring tool in the main task bar of Maya. User could open up the details panel by click the menu item to create a new task.

The following scratch shows the GUI layout of details panel of the tool:



- Facade layout section:
  - Dropdown menu for selecting facade layout from our database
  - File selector for importing user-defined custom layout.
- Component sections:

`

    - Dropdown menu bar for selecting window texture and models. Selected model will be showing using a 2d texture
    - Dropdown menu bar for selecting door texture and models. Selected model will be showing using a 2d texture
    - There might be more components apart from windows and doors which might be added to this section later in the implementation
- Grammar section:
    - A text field showing the extracted grammars from the input layout
- Tuning section:
    - This section consists of a set of sliders and input fields where user could tune some parameters of the generated grammar to customize the building
- Execution section:
    - Generate grammar button: This button is to generate split grammar of the input facade layout, and this button should be activated only when there is a valid facade is selected. Once clicked, the tool should generate the grammar and display the grammar in grammar text field
    - Generate model button: This button is to generate building models using the split grammar. Once clicked, the generated model should be shown in the viewport of Maya. Once the user changes the parameter settings of the grammar, they need to click this button again to regenerate model.

1.4.2 User Tasks

The user tasks includes:

- User could select the preset facade layouts using the dropdown menu in the defials panel
- User could upload their own facade layout by using the file browser widget
- User could select different types of window and door model to use in the building model
- User can tune parameters of the generated grammar using slide bars and input field in the tuning section of details panel to produce variations of the building

In order to produce variations of facade, user need to have a sense of how each parameter could influence the overall looking and structure of the facade. Of course user could play with those parameters to explore more possibilities

1.4.3 Work Flow

- Describe the work flow in terms of how the tool is used (i.e. steps, order of operations, etc.) within the Maya or Houdini authoring environment.
- Provide an example of a typical session.
- How does this work flow solve the problem or meet the production/development need.
- What does the tool require as input and what does it produce as output?

The typical work flow of using this tool would be like this:

`

- User open up Maya, and make sure our authoring tool plugin is loaded
- User click on the menu item of our tool, and create a new session
- On the right side panel of Maya, the details panel of our plugin show appear automatically
- User need to choose one of the facade layout from our preset database or import their own facade layout using the file browser widget
- Then user need to choose one model/texture for the door and one model/texture for the window from the component widgets section in the details panel.
- After that, user could go to the bottom of the details panel and click generate grammar button to generate split grammar of the input facade layout.
- After the processing is done, the generated split grammar should be displayed in the grammar text field.
- User could then click the generate model button at the bottom to generate the building model using those grammar.
- User could modify the grammar directly inside the grammar text field, and then click the generate model button again to apply the changes.
- User could also adjust some parameters of the grammar in the parameter section and then click the generate model button again to apply those changes.

The input of this system is a 2D facade layout and user specified component models. The user can also interact with the system by adjusting some parameters of the split grammar. The output of the tool is procedurally generated building model with facade that looks similar to the input facade layout. The overall work flow is quite simple and straightforward, and should address the needs of artists that quickly and efficiently generate a bunch of natural looking models.

2. AUTHORING TOOL DEVELOPMENT

2.1. Technical Approach

2.1.1 Algorithm Details

The main goal of the algorithm is to produce a grammar that reproduces a given facade. This works by adding rules to the existing grammar until the features in the image are reproduced. Potentially many grammar rules could generate the same facade, so the goal of the paper is to produce the simplest set.

We will start with a parsed set of facades. These are annotated facades that are available on several websites such as CMP Facade Database, eTRIMS image Data Base. The authors also have a database that we will use. We then will find any set of rules that reproduces the facade. At first the set of rules could be long.

Following the paper we will search for the minimal set of rules to produce the given facade. To find the simplest set we will start by adding a cost function to each additional rule. We will

`

search for a sets of rules that randomly pick split and repeat rules and chose the rule set with the lowest cost. Following this we will implement parts of the authors' algorithms.

Because the space of possible rules is so large the authors implement an approximate dynamic programming approach to selecting rules. The first part of that is a top down approach. This associates a cost with each additional rule and then determines a value function based on future values. The values are solved recursively, This approach is the approximate dynamics approach from Powell, Warren B. *Approximate Dynamic Programming: Solving the Curses of Dimensionality*. Wiley-Blackwell, 2011.

To get better rules closer to the real facade we will also implement a bottom up approach. This approach generates a heuristic; the segmented facade and looks for repeated patterns like windows that repeat. The algorithm then exhaustively searches for as many repeats as possible. This leads to a rule that describes more features more concisely and so will have a lower cost. Similarly there is a splitting Heuristic in the paper that also describes large scale features. We will implement these algorithms because they both are easy to understand and lead to concise rule sets.

The authors have a set of facades annotated facades and the grammar produced. We will use those facades and grammars to test our implementation. Our assumption is that the set of facades employed covers the range of problems from parsing facades.

2.1.2 Maya Interface and Integration

The main algorithm will be implemented in C++ outside of Maya or Houdini. This will take parsed facades and generate grammars. Maya will be used to present the user with options. The user will at first chose which facade to use as a basis. This will call our C++ plugin that will actually generate the grammar. The grammar will then be used back in Maya to generate a facade from the grammar.

In the simplest form the interface would take the parsed facade image with textures for all the distinct elements. The user would select a facade and a new facade dimension, and the algorithm will produce the new facade.

We will use C++ to generate the grammar from the input facade. We will use another C++ module to take the grammar and generate the new facade.
MEL will be used to build the GUI and user interface. MEL will also be used to render the final model facade.

In the C++ there will be a list of facade elements like windows and doors. These will be labeled with an integer for each and stored in a table. We will label the structures as Terminal or nonTerminal and subject to further division. The generated grammar will be applied to the facade.

`

2.1.3 Software Design and Development

There are two main custom nodes. The first custom node would be to take a facade and generate a minimal grammar. The second custom node will take the grammar and generate a similar facade. The first Node will need to parse the annotated image for features and locations. This image will then be segmented and rules will be proposed that model each segment. The segments will be further subdivided and grammar added until the facade can be solved. The rules will be stored as strings in a table. The second custom node will take the rules and procedurally build the facade. This will be a custom node but will not need many special data structures because once a command is issued in Maya, the data will be stored there.

2.2. Target Platforms

2.2.1 Hardware

The requirements are simply the requirements for running Maya and Visual Studio 2015. Ideally we would have 8 GBytes of RAM, and a 64 bit machine running Windows or Linux.

2.2.2 Software

We will use Windows 10, Maya 2018, and the Microsoft Visual Studio 2015 compiler. Prior versions will also work.

2.3. Software Versions

2.3.1 Alpha Version Features (first prototype)

- Input a parsed image with labelled sections and produce a grammar of rules. The grammar may not be minimal.
- Have a GUI that accepts as input the type of facade and the size of the desired image. This version may only use textures for doors and windows.
- Write the Second Node that will take the grammar and produce the facade.

Important Milestones

- Write code to accept the parsed facade and store the data in the Nodes data bases.
- Generate a grammar that explains or predicts the test facade.
- The demo will be showing some model facades as input and the predicted facades.

2.3.2 Beta Version Features

- Improved grammar parsing to make a more minimal set of rules. Grammar that looks at heuristics in the image such as repeated windows and features and extracts larger more succinct rules.

`

- Add models for the windows, doors, etc. This will result in more realistic looking facades.
- the demo/will allow the user to pick a facade a window type and a door type and make more realistic facades with user selected components.

2.3.3 Description of any demos or tutorials

- The tutorial will be a screen recording of using the GUI. That will take the user from selecting facades, window styles, door styles etc and producing a facade or a simple building.

`

3. WORK PLAN

3.1. Tasks and Subtasks

Task 1 – Generate split grammar from input facade layout

Duration: 2 weeks

For this task, what we need to do is to build a C++ program which takes an input of facade layout 2D image, parse it, and run the buttom-up and top-down algorithm to generate the split grammar that best describes the facade. Finally the output of the program should be a set of split grammar strings and be the input of our procedural modeling stage

- ***Subtask 1.1. Parse input facade layout***

Write a C++ program that parse the 2d input facade layout and encode the layout structure, terminal and non-terminal regions, and store all information in data structure for later use.

- ***Subtask 1.2. Implement bottom-up algorithm***

Write a C++ program that implements the bottom-up algorithm described in the paper which groups regions of the facade layout while maximizing translational symmetry of grouped regions. The bottom-up analysis should do the symmetry detection as well

- ***Subtask 1.3. Implement top-down algorithm***

Write a C++ program that implements the bottom-up algorithm described in the paper which splits the regions using dynamic programming formulation. To split the region we need a way for selecting rules, and this is done by using a splitting heuristic.

Task 2 – Generate model using split grammar

Duration: 1 week

What needed to be done for this task is that we need to take the generated split grammar string comes out from last taks and generate procedural model from it. We need to build a system that parse the split grammar and draw primitive geometries inside the Maya viewport. We also need to create interfaces that allows the user to manipulate several parameters of the system.

- ***Subtask 2.1. Build a system to parse split grammar***

We need to build a C++ program that parse the split grammar and draw geometries to the viewport according to the split grammar. The final output of this system should be a complete building model draw from the split grammar

`

- ***Subtask 2.2. Create interfaces for parameter tuning***

Create interfaces that allows the user to change the parameters of the building such as the size of components. Link the interface with the Maya GUI so that the user can adjust the model through Maya interface

Task 3 – Generate assets and final integration

Duration: 1 week

We need to generate a set of preset assets for window/door models that can be used by the user. After that we need to integrate the our system into maya by transfer our C++ application into maya plugin. Finally we need to do so tuning and final debugging

- ***Subtask 2.1. Create assets***

We need to model/download some window/door/… assets that the user could use. If we have time we should also implement the facility of allowing user to import their own model assets

- ***Subtask 2.2. Integrating into Maya***

In this task the main thing needed to be done is to transfer our C++ application to Mata plugin. This involves writing the MEL script for GUI, and creating custom node to integrate our system.

3.2. Milestones

3.2.1 Alpha Version

List the tasks and subtasks that must be completed for the alpha version.

- Parse input facade layout
- Implement bottom-up algorithm
- Implement top-down algorithm
- Build a system to parse split grammar

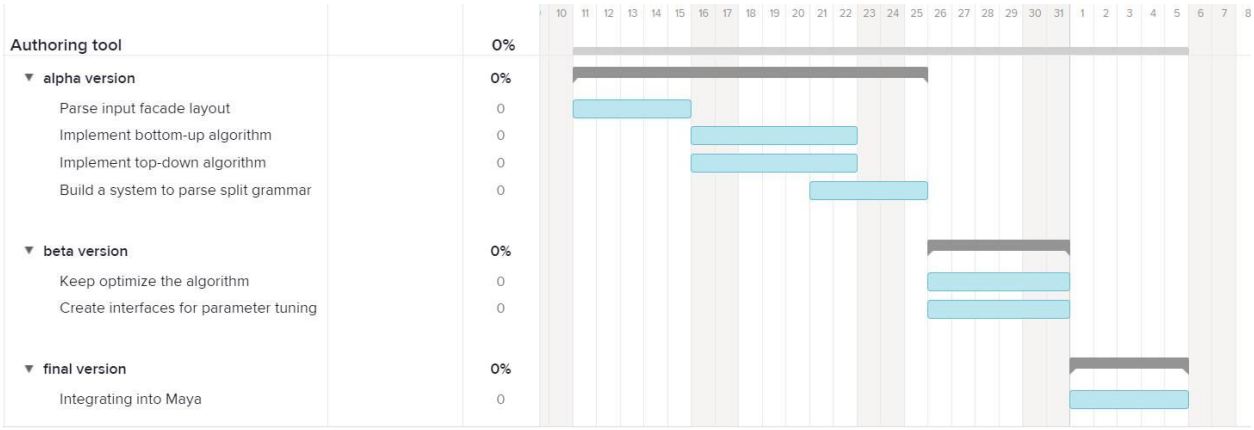
3.2.2 Beta Version

- Keep optimize the algorithm
- Create interfaces for parameter tuning
- Integrating into Maya

3.3. Schedule

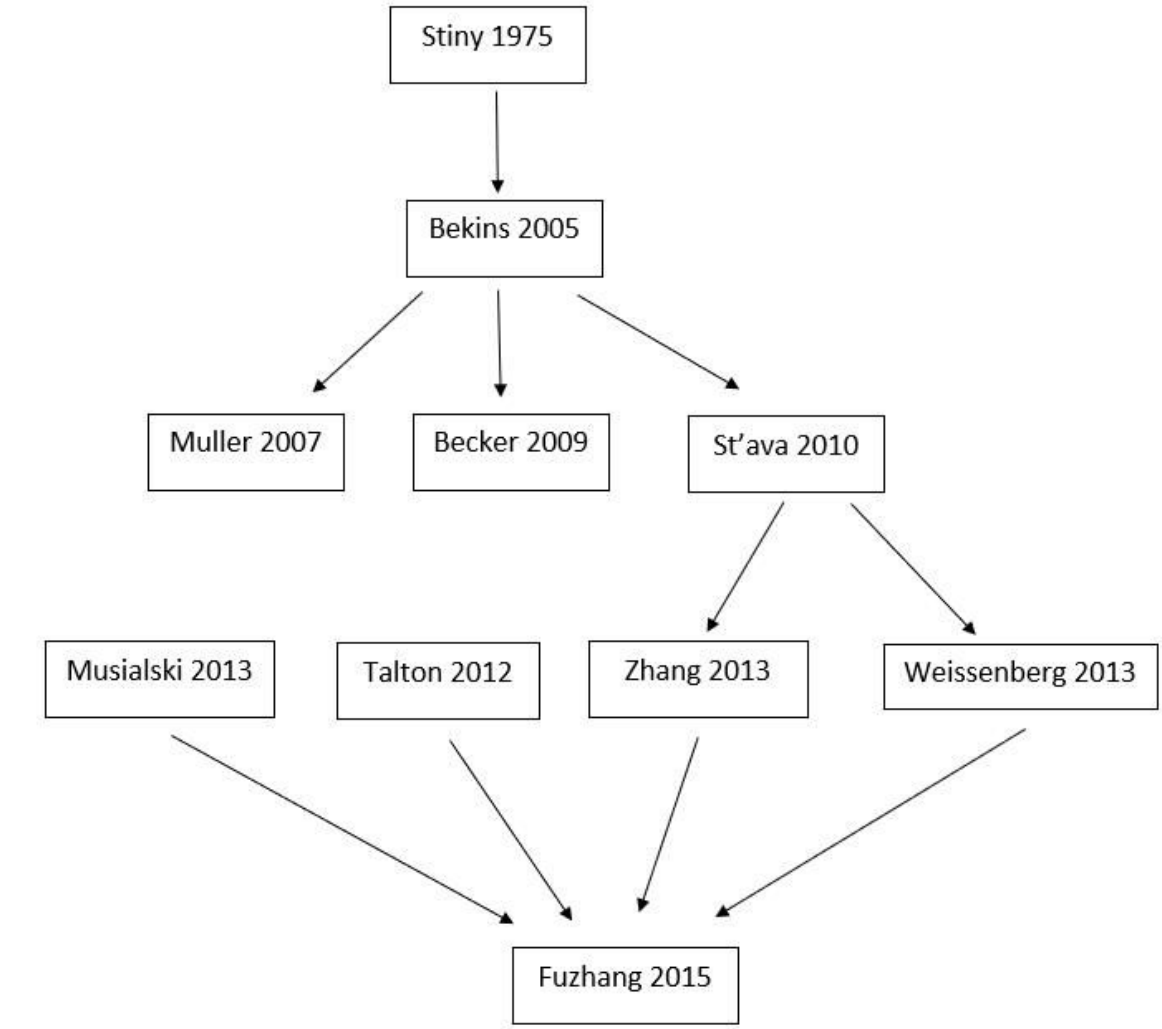
`

Example Gantt Chart



`

4. RELATED RESEARCH



[Stiny 1975] Pictorial and Formal Aspects of Shape and Shape Gram-mars: Stiny introduced the seminal idea of shape grammars as a formal approach to architectural design. Shape grammars were successfully used for the construction and analysis of architectural design

[Bekins 2005] Build-by-Number: Rearranging the Real World to Visualize Novel Architectural Spaces: This paper proposes a grammar that splits a facade into floors and then encodes a

`

one-dimensional sequence of elements. The advantage of this approach is that it reduces the problem to a sequence of one-dimensional problems, but the disadvantage is that it only applies to facades with this structure and is not applicable to general two-dimensional layouts. This approach is therefore more similar to finding the parameters of a pre-determined shape grammar

[Muller 2007] Image-based Procedural Modeling of Facades: This paper combines the procedural modeling pipeline of shape grammars with image analysis to derive a meaningful hierarchical facade subdivision.

[Becker 2009] Generation and application of rules for quality dependent façade reconstruction: This paper integrates automatically inferred rules into a data driven reconstruction process. Dominant or repetitive features and regularities as well as their hierarchical relationship are detected from the modelled façade elements and automatically translated into rules. These rules together with the 3D representations of the modelled façade elements constitute a formal grammar. It holds all the information which is necessary to reconstruct façades in the style of the given building.

[St'ava 2010] Inverse procedural modeling by automatic generation of l-systems: This paper is the first formal treatment of the inverse procedural modeling problem in computer graphics. The authors present an important step towards the solution of the problem of inverse procedural modeling by generating parametric context-free L-systems that

`

represent an input 2D model. The algorithm takes as input a 2D vector image that is composed of atomic elements, such as curves and poly-lines. Similar elements are recognized and assigned terminal symbols of an L-system alphabet. The terminal symbols' position and orientation are pair-wise compared and the transformations are stored as points in multiple 4D transformation spaces.

[Weissenberg 2013] Procedural 3D building reconstruction using shape gram-mars and detectors: This paper proposed approach that complements Structure-from-Motion and image-based analysis with a 'inverse' procedural modeling strategy. The authors reconstruct complete buildings as procedural models using template shape grammars. In the reconstruction process, they let the grammar interpreter automatically decide on which step to take next. The process can be seen as instantiating the template by determining the correct grammar parameters. This algorithm works very nicely for simple facades

[Zhang 2013] Layered analysis of irregular facades via symmetry maximization: This paper presents an algorithm for hierarchical and layered analysis of irregular facades, seeking a high-level understanding of facade structures. By introducing layering into the analysis, we no longer view a facade as a flat structure, but allow it to be structurally separated into depth layers, enabling more compact and natural interpretations of building facades. While their idea to structure a facade is excellent, the actual algorithm to obtain the subdivision leaves room for improvement

`

[Talton 2012] Learning design patterns with bayesian grammar induction: This paper describes an algorithmic method for learning design patterns directly from data using techniques from natural language processing and structured concept learning. Given a set of labeled, hierarchical designs as input, they induce a probabilistic formal grammar over these exemplars. Once learned, this grammar encodes a set of generative rules for the class of designs, which can be sampled to synthesize novel artifacts. Worth to notice is that Martinovic et al. propose a heuristic to extract a split grammar from a facade layout.

[Musialski 2013] A survey of urban reconstruction. Computer Graphics Forum: This paper provides a comprehensive overview of urban reconstruction. The work reviewed in this survey stems from the following three research communities: computer graphics, computer vision, and photogrammetry and remote sensing.

[Fuzhang 2015] Inverse Procedural Modeling of Façade Layouts: This paper defines the criteria of "good split grammar" which is that the split grammar should be semantically meaningful and that the procedural description should encode identical regions consistent. To better define a "good grammar", this paper used a fundamental observation that a shorter description is usually preferable to a longer one. To tackle the problem, the authors use two components. First, they propose a cost function to evaluate how meaningful a grammar is. Second, they propose an optimization framework based on approximate dynamic programming to extract a grammar that minimizes this cost function

`

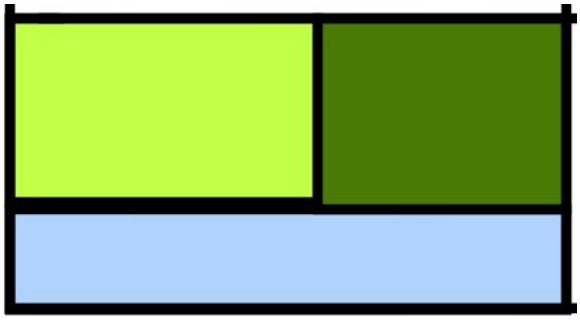
5. Results

5.0 Documentation

The algorithm to Parse Layouts.

Creating the algorithm to generate the grammar as stated earlier consisted of constructing a list of all possible groups organized by location, and using that information to generate candidate split rules and associate with them a cost. With a good estimate of the cost of a split rule, one can limit the splits explored to those that minimize cost. It is crucial to limit the number of possible splits because for each non-terminal rectangle with three split lines, there would be 8 possible splits. The number of possible split rules grows exponentially with each subdivided rectangle.

Here the approach we used was to develop a series of structures and functions to retrieve all non-terminal groups. Terminal groups are groups with a single name and are indivisible, whereas non-terminal groups are rectangles composed of terminals. We are most interested in repeated groups because those could lead to a repeat rule which would allow stretching of facades where sub-structures would repeat.



A non-terminal group node consisting of 3 terminal groups.

The bottom-up structures create all nonterminal groups, organized by location and unique id. The top-down uses that to generate grammars. The bottom up structures and functions that we implemented made it simple to generate and implement a thorough top-down. Our generated grammar should find rules that are simple and prefer repeats. Simple allows for the grammar to generalize well so that different shape facades maintain the same look. A grammar that uses many repeats allows for stretching that increment the number of elements, minimizing the need to stretch terminals.

5.0.1 Data structures Overview

`

As we generate rules to split a given facade with grammar rules, we kept track of all non-terminal groups. To find repeated groups, we first create a single group, and checked whether this group wos the same type as a prior group. All distinct groups really were distinct. At a given location, and for a given rectangle at that location, one wants to know all possible ways of dividing it to find the simplest rule to divide it. We developed a structure that returns all rectangles of a given width at every facade location. Here single groups also need to be kept track of even if not repeated because they suggest ways to cut a facade. Any line that splits a rectangular region of a facade, may split repeated groups rendering them no longer useful. Thus for every split line, we need to know how many repeated groups would be no longer usable. To keep track of single and repeated groups the data structures must support adding and removing of groups.

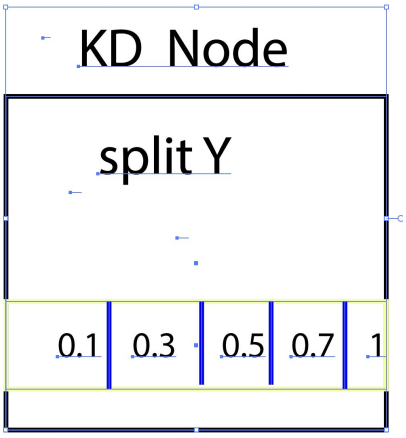
There were several data structures made in order to create the bottom up structure. In order to navigate locations in the facade, we often need to add floating point numbers together to get new locations. Since locations are used as keys, we need a good way to test if two locations are the same or not, regardless of floating point error. A.K. wrote previously classes Efloat and EVector that store a float and its roundoff error. When Efloats are added, the actual result is within the stored error, so that Efloats can be tested for equality consistently. These were used throughout the Bottom up to allow floating point numbers to be keys in maps.

5.0.2 KD Tree

The first is a KD tree that follows the inputted xml file that stores the inputted facades. Each Node provides a size and bounding box, a list of children, a split direction and splitlines. as well as consistency and error checking flags. A utility called TINYXML2 written by Lee Thomason was used to parse the file. Once parsed, a KD tree like structure was made that has the same divisions and children that the xml file had.

The KD spatial tree that follows the xml input file and first splits the facade along X or Y into multiple children, and in turn spits the children in X or Y, until the tree is parsed. The terminals are leafNodes and have names from the xmlfile and the nonterminals are BranchNodes called unlabelled. Nodes store a count of the number of non-terminal nodes stored within it. The structure does not change after reading in the xmlfile.

`



KD tree Node

The nodes in the KD tree have a NodeValue:

```
struct NodeValue {
        uIDType         uid;  // unique id of the node
        std::string     name;  //name of the node
        unsigned        n;  // number of terminals in this node Term = 1
        bool terminal() const;  // terminal there is only one terminal Node here.
};
```

which has a unique id number which characterizes the node, and a unique name. NodeValues are shared, so each node with the same unique id shares the same NodeValue.

The Nodes hold a NodeValue plus the structures needed to navigate the KD tree.

```
struct Node {
        Node(const EVector& sz, const EVector::Axis sd, std::vector<Efloat>&& ss,
                std::weak_ptr<const Node> p,
                        std::shared_ptr<NodeValue> v);
        std::shared_ptr<NodeValue>  v; // the potentially repeated structure
                                       // stores all the information of the node
        bool terminal() const;  // use v's terminal
        const EVector       size; // holds the size of the box
        const EVector::Axis splitDir; // split along x or y
        std::vector<Efloat> splits;// the location of the splits
        std::vector<std::shared_ptr<const Node>>  children;
        std::weak_ptr<const Node>  parent;
        virtual ~Node() {}
};
```

Nodes hold the nodeValue plus all the information needed to navigate the KD tree: the size, the splits, the split directions, the children and the parent. Here we note that Efloats and EVectors so

`

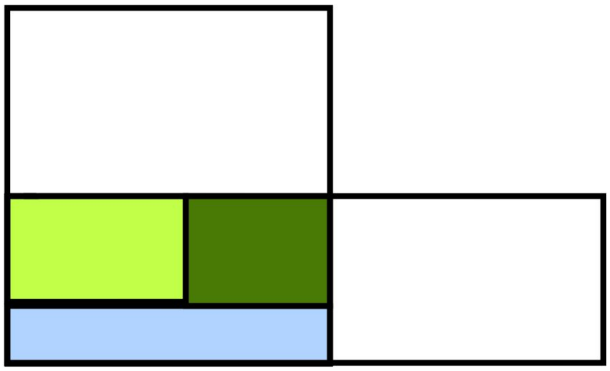
that equality tests work even with roundoff associated with floating point addition. The parent is weak to allow for automatic destruction.

Because of the KD tree organization, we just need the splits (Efloat) and an initial location(EVector) and the node Size (EVector). With that, the location of every branch Node was calculated. This uses less space than storing in each Node, the minimum and maximum bounding box dimensions, which are both redundant. The paper authors do not use Efloats and instead store the size, min and max dimensions of each box. To avoid roundoff, they must have had to avoid addition completely and hope that the min of one box matched exactly to the neighbor max. Here the code is more robust, uses less memory and is easier to follow.

The Node has mostly constant elements and in addition it is accessed by a shared pointer to a const Node. This is because once a node is made and a uid established for it, it does not change.

5.0.3 LL Corner

The next structure called LL stores non-terminal groups by lower left corner. That is the location in the Facade where the group is found. All non-terminal nodes that have the same lower left corner are stored in the same map. This is needed in the GroupMap function below, so that one knows what groups are available at neighboring locations available to form other groups. As described below, we look to the neighbor location for a group that matches a given groups X or Y dimension and use that to form a new group. This is also used centrally in the top down algorithm to find all non-terminal rectangles that split a group entirely either along X or Y.
LL Corner Supports insertions and deletions.



\

In this example shows 3 non-terminal and one terminal region in the same lower left corner.

The lower left structure is derived from the Node structure above:

```
        struct LeafNode :Node {
                        LeafNode(const EVector& sz, const EVector::Axis sd,
std::vector<Efloat>&& ss,
                                                std::weak_ptr<const Node> p,
                                                std::shared_ptr<NodeValue> v);
                                // stores all the groups organized by lower left corner
```


`

```
                        mutable XYWidth LL;
/*************************************************************************
*****************************
 * @func      addGroupToXYLocMap.
 * @args[in]  std::shared_ptr<const Node> groupNode
 * @return[out]  InsertType tells if there was a matching Node there or not.
 *              Whether to insert the new Node
 * @brief     will add an entry to the map or create an entry if need be.
 *                returns true if the add was successful.  returns false if there
 *                is already a group of the same X, Y width that has not expired
 *                (meaning do not add).
 *                There should only be one group with the same XYWidth.
 *                If the new group matches the old one, this will keep the old.
 *                The prior does not have to be eliminated with a call because if
 *                it is deleted in the GroupMap it will be expired here.
 *
 *
*************************************************************************
**********************/
                        InsertType addGroupToXYLocMap(std::shared_ptr<const Node> inNode)
const;
/*************************************************************************
************************
 * bool removeFromXYLocMap will remove a Node from the XY map.  It should be
 * found.  returns true if found and removed successfully */
                        bool removeFromXYLocMap(std::shared_ptr<const Node> inNode) const;
/*************************************************************************
**********************
 * @function  list<std::shared_ptr<const Node>>  findXYLocMap(EVector::Axis ax, Efloat
 *                size)
 * @params[in]    ax is either X or Y, the axis to look for;
 *              size is the size to match to,
 *              if n is there then
 *              n    is the number of terminals to match to
 * @params[out]   list<std::weak_ptr<const Node>>  the list of weak_ptrs to
 * Nodes that match.
 *
*************************************************************************
**********************/
                std::list<std::shared_ptr<const Node>>  findXYLocMap(EVector::Axis ax, Efloat
width, unsigned n) const;
                std::list<std::shared_ptr<const Node>>  findXYLocMap(EVector::Axis ax, Efloat
width) const;
        };
```


`

The first structure to note is the mutable XYWidth LL. It is mutable because all the other elements of the Nodes never change, so this way the Node used for navigating is constant. This is a map that holds Nodes af all groups that share the same location. The map is a map of maps that first uses X Width as an Efloat to find a map of nodes that share that width. Those nodes are organized in another map using the Efloat width y as a key. We used a map which is a binary tree as opposed to a hash table because the key (Efloat) supports the less than operator. The hash table does not support the less than operator and also the hash table requires a hash function that supports equality. The hash function should map two efloats to the same bin even if they are not exactly the same and that is not trivial. To avoid that, a map uses the less than operator, supports equality, and accepts Efloat keys.

findXYLocMap produces a list of all Nodes at a location that have the same width. This is critical both for building up the Group Maps below and the Top down. Pass this function the axis you want, the Nodes width and optionally the number of terminals in the node, and it will produce a list of all matching Nodes that all have the same width specified.

If you specify the X width and the Y Width, there is only one group that could be in that spot because that corresponds to a rectangle in the facade. That group could be built in different ways but it is still the same group.

For GroupMap below this is used to match neighboring groups as described below. For the top down, this finds all the rectangles or groups that cut a given group across all X or all Y, giving the user the list of possible choices to partition a facade section.

To find locations in the facade we chose to overload the LeafNode in the KD tree and navigate the tree. There was another option to use another map call like the XYWidth structure above and store the LL Corner structure in that map. That map would not be more efficient than the KD tree and would certainly be more complex as a structure (even if part of the standard Template library), so we opted to use the KD tree for navigation and develop our own navigation functions.

To navigate the KD tree the routines were;

```
/**************************************************************************
*********************
 * @func    std::shared_ptr<Node> findLLNode(std::shared_ptr<Node> init, Evector&
lowerLeft,
 *                                  const EVector& term)
 * @params[in]    std::shared_ptr<const Node> init: start Node. If it is a terminal node then you
 *                        can navigate the location KD tree and go up and down.
 *                        If it is a groupNode in the KD tree then you also can go
 *                        up and down.  It can also be used to find the Lower Left
 *                        corner of a group node. In this case the lowerleft and
 *                        term should be the same.
 *             EVector& ll init location of that node in the
```


`

```
 *                          spatial structure. This will get updated for each call up and down, so
 *                          will get modified
 *                           as the algorithm traverses the tree.
 *                  const EVector&   term the lower left corner that one wants to get to
 * @params[out]   std::shared_ptr< const Node>  the terminal node with this coordinate or null if
no
 *                   such pointer exists
 * @precondition  initial node exists.
 * @brief         will look through the tree starting at the GroupPair, searching up the tree
 *                   or down the tree and return the node that has the lowerLeft corner at term

**********************************************************************************
******************/
        std::shared_ptr<const LeafNode> findLLNode(std::shared_ptr<const Node> init,
EVector& ll,
                                    const EVector& term);

// provide a child and an absolute LL coordinate, and this finds the lower left
// coordinate of the parent.
        void   parentLLCorner(std::shared_ptr<const  Node> parent, std::shared_ptr<const
Node> child,
                                     EVector& minValueChild);
```

findLLNode was used at least twice. First, given a group Node and its location we would find the lower left corner. The original group Node may be made by the Group Map and may not be in the KD tree, but the lower left corner would be there. With that corner one could navigate the KD tree. The next use was given a LL corner, it can find a Terminal location anywhere else in the KD tree. It will go either up the tree or down. To find a groups neighbor, the LL corner is probably close to the neighbor so this routine is fast. It is better than starting at the root node each time which could require many evaluations.

A helper function is parentLLCorner which returns the parent LL corner given the child LL corner. In linearly searches over each child in the parent for a match to the child.

LeafNode supports adding and deleting non-terminal (NT) groups. and that is necessary as we see below

5.0.4 The Group Map and Names.

The next structure is an indexed-based hashtable called a GroupMap, that organizes terminal and non-terminal groups by a unique id. For each group, it stores a GroupPair which consists of a Node and a lower left location. It supports insertions as new nonterminal groups are added and

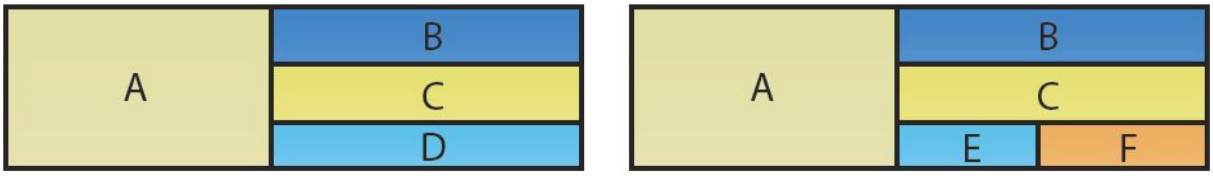
`

also deletions when the generated grammar rules imply that some of the nonterminal groups would never be used.

The next structure is a names hashtable that associates a string as a key with each unique id and is used for debugging and identifying groups, but not strictly necessary for generating grammars.

The GroupMap is initialized by storing all the terminal nodes in the GroupMap. For each repeated element we need to store both the location and the Node because groups with the same name may have much different sizes. For example in Layout.xml, "wall1" can span the entire facade width or else be ¼ of the width. Thus the size does not characterize the group. Rather, a group is identified by its subgroups, and the arrangement of terminals (X or Y). For example, a unique group would be A and B to the left and that would be distinct from A and B stacked vertically. Since size is a property of individual groups, and is not the same for all groups with the same uid, size is stored in the Node (not NodeValue). In addition, every a unique group is a Node and location; the node described above determines the locations of the subgroups.

To build up all the unique groups it is clear we need to build first all groups with lower number of terminals before one can build up groups with higher numbers. In the example below, if BCD were not a group already, then A on the left would find no group to the right that matches its vertical height. Similarly, before C, E, F form a group, E and F must already be a group.



The routine then finds all non-terminal groups. For each group (terminal or non-terminal), go to its lower left node using *findLLNode* (described in LL Corner).
Use the same function and the groups size to go the the lower left corner of the terminal node at the next corner. For example in the picture above *findLLNode* can go from the LL corner of A to the LL corner of D. It then uses *findXYLocMap* to find all prior groups that match A both with the width and the number of terminals. In the left example, to make a non-terminal group of 4 nodes, we would go to A's right bottom corner (with *findLLNode*) and D's LL corner. We then would use *findXYLocMap* with Y width matching A's and 3 terminals. In the right example we would look for 4 terminal matches. In this way all the group nodes can be built.

To build a group from simpler groups, we find two matching groups and make one parent to that that holds the entire size (called *makeParentGroup*). This is stored then both in GroupMap and in the appropriate LL corner. When a new group is made, a check is done to see whether there is a previous group with the same structure (sameGroup), that is with the same exact children. If there was, the old group name is used for this one and the number of repeats increased. This group gets a name that includes the new group number, the group numbers of the two children and the split Direction. This information is common to all nodes of the same type.

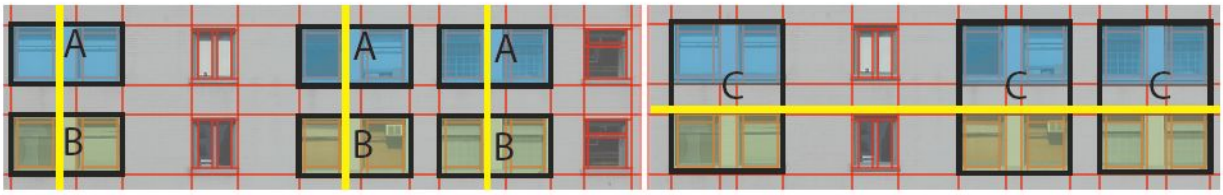
`

For complex groups, there is more than one way to make the exact same group. For example on the left, the group of BCD could be made of BC and D or B and CD. These same groups should be counted and identified as the same in order to have an accurate record of the repeats there. If a group was created before, it would be found in the same location with the same X, Y width. *findXYLocMap* is used to find these repeats and then reliable the group to be included with the prior.

To get all NT groups one only needs to organize the groups by lower left corner. For example if F is to the right of E then the group E F will be found looking for matches to E on the left. F E if any exist, is searched for by looking for groups to the left with F as the search group. Thus one just has to look to the left and above for all groups, and to the right and bottom will be found when matches to the other group are searched for. The same rule, simply looking to the left, properly accounts for groups like A A, counting them just once.

It is necessary to have the Group Map structure so that one can sequentially look for groups of a given size, trying all possible choices. Even if an NT group is not repeated. it is stored in the GroupMap structure and the LL Corner structure. This allows finding all lines that split a given node, both in X or Y regardless of whether the KD Node has X splits or Y splits. It is described in the Discussion and below.

5.0.5 SplitCost Data structure



For a given rectangular region in a facade, one needs to choose which lines to use to divide it. In this example there are divisions along both X and Y. With a series of lines, one has to choose which ones to pick for splitting rules. To help determine the line, we associate a cost to that line by counting nonTerminal groups that are split by that line and therefore not accessible anymore. The paper suggests associating a cost of each line by counting the terminal groups in each repeated nonterminal node that is split by the line. In the example to the left each line would incur a cost proportional to the number of terminals in A + number of terminals in B.

The flow here is to first store with each line in the KD tree all the nonterminal groups that are split by that line. for each repeated NT group, we navigate the KD tree and add it to the split line in the branch node that split it. When a line is chosen, the cost of using that is calculated by grouping together all nodes that are split by the line and counting terminals included. This will

`

suggest which lines incur the least cost and which ones to choose for splitting rules. Once a line is chosen, those NT groups should be removed from all structures (SplitCost, LLCorner, and GroupMap).

Once removed there may be no repeats of that type left, one left, or more than one left. If there is one then that node should not be considered a repeated split. anymore and so it should be removed from the SplitCost structure. This keeps the accounting straight, so that each new line only is charged for the new groups that it is rendering unusable.

Initially the NT group is added to all split lines that break it, and these are removed once the NT group is no longer usable.

To support this the BranchNodes in the KD tree were overloaded to hold the Bnanch Structure.

```
        struct BranchNode :Node {
                        BranchNode(const EVector& sz, const EVector::Axis sd,
std::vector<Efloat>&& ss,
                                        std::weak_ptr<const Node> p,
                                        std::shared_ptr<NodeValue> v);

                        mutable std::vector<WeakMap> splitGroups;
                        // adds a weak reference to the groupNode split by the
                        // line.  LL corner is not needed so not stored
                        // throws exception if failed to insert
                        void addGroup(std::shared_ptr<const Node> NTGroup, SplitItPair) const;
                        // removes a group from the branchNode
                        // true if found /false if not found; throws exception
                        // if expired
                        bool removeGroup(std::shared_ptr<const Node> NTGroup,
std::vector<int>::size_type ) const;
        };
```

It has a mutable vector of WeakMaps, one for each splitline. Each WeakMap holds all uid keys, and Nodes of all NT groups split by that line. It stores the nodes as weak smart pointers. There is an add group and remove group.

To find all the lines that split a group a function used is:

```
/*************************************************************************
*********************************************
 * @func      addNTGroupToSplitLines will insert a non termial group into the
 *                spatial structure of nodes including a weak reference in all
 *                split lines in all branch nodes that split the group.
 * @params[in]   GroupPair LL, the starting search direction
 *             GroupPair NT, the nonTerminal Group Node
 *            An important observation is that the splitlines that divide groups are never
```


`

```
*                on the bounding box, they are always in the split segments within it.
*                Furthermore for every line that splits groups, it overlaps with only one internal
splitLine in the parent and
*                many as bounding box edges in the children.  The NT Groups are
*                stored once in the internal segments and not on the edges.
 *
**********************************************************************************
**********************************/
        void addNTGroupToSplitLines(GroupPair ll, GroupPair ntGroup);
/**********************************************************************************
*********************************************
 * @func      removeNTGroupFromSplitLines will remove  non termial group from all the split
lines.
 *                in all branch nodes that split the group.
 * @params[in]   GroupPair LL, the starting search direction.  Must be a
 *                        GroupPair of a terminal group.
 *            GroupPair NT, the nonTerminal Group Node
 *
**********************************************************************************
**********************************/
        void removeNTGroupFromSplitLines(GroupPair ll, GroupPair ntGroup);
```

These take a NT group (Node, location) and a starting location in the KD tree LL, and add the node to all the KD splits that would cut the Node. To find all the lines it uses a class called *LineIntersects* derived from *LineBasics* (in the cpp file). Those classes have functions that determine whether a given KD node, say B, overlaps with the NT node, in which case in needs to be searched. Another function is *groupWithinLocation* which determines if an NTGroup is entirely within the KD Node B. If it is not then the parent needs to be considered. These two functions allow us to get the parent KD node that entirely includes the NT Node. Once found, one navigates down to get all the children that overlap.

One function *LineIntersects*(Efloat) takes a splitline from Node B and determines if the NTGroup is split by that line. For a line to split it, the line must not be on the border but fall within the NTGroup. Another *LineIntersects*(std::pair<Efloat, Efloat>) determines if a child of B overlaps with the NT Group and if so, explores the child.

Critical here is the use of Efloats because that distinguishes whether an NT group borders a line or is split by it. Efloats perform this task naturally without worying about roundoff.

It is also necessary to retrieve all NT Nodes split by a line even if the line spans multiple KD nodes. The yellow lines in the above figure would span multiple KD nodes. That is done as follows:

```
LineSegment(GroupPair gpr,  SplitIt  split);
const EVector::Axis  ax;  // the direction of min, max, say Y
const minMaxPr  pr; // min and max values of the line say ymin, ymax.
```


`

```
                const Efloat transverseVal; // the one transverse value say x.
        };

/*****************************************************************************
*********************************************
 * @func       allSplitGroups will take lineSegment and return all the groups
 *                      that are split by the line segment. The groups returned
 *                      are unique but there may be more than one ntGroup with
 *                      the same index.
 * @params[in]   GroupPair LL, the starting search location.  Will go up and
 *                      down the tree to find all overlapping lineSegments.
 *           lineSegment & line , the line segment that defines the line you
 *           are looking for.
 *  @return      a map of const Nodes that are cut by the line
 *  @brief      There may be a single line if the line happens to be exactly
 *                along the kd tree splitlines, or it could be along multiple kd
 *                tree splitlines.  This gathers the splits from all of them. It
 *                does not remove any groups from branch Nodes.
 *
*****************************************************************************
***********************************/
        NodeMap allSplitGroups(GroupPair ll, LineSegment& line);
```

The line segment is just a vertical or horizontal pair of points that can encode any line that splits a rectangle. *allSplitGroups* then first finds the parent Node that includes the split line, then finds all line segments that align with the given line. All the nodes are added to one NodeMap where duplicates are removed. It uses a derivative of *LineBasics* to implement the same functions. Templated versions of intermediate functions are used here and as described for *LineIntersects*. Templates allowed code debugged for one to be used for all and avoids duplicates.

5.0.6 Copying and Removing Nodes.

To generate a set of grammar rules initially, the structure BottomUp is created, which parses the XML files, creates the KD tree, the GroupMap, the LL Corner structures and the SplitCost data structure. As grammar rules are created and smaller rectangles are parsed, the NT Nodes not selected must be removed to keep the SplitCost accurate.

The main structures are stored in another structure called BottomUp. That has the following constructors:

```
                // parse the XML Document
                // by first opening the file
                BottomUp( const char *);
```


`

```
// this allows one to copy a BottomUp structure.  The copy does
// not refer to any nodes in the original so original can be
// changed or deleted and copy remains intact.  This allows one
// to delete split groups in copy without affecting original.
BottomUp( const BottomUp& );
```

The first takes an xml file and does the parsing as described above. The second takes an already created BottomUp structure and copies everything over. Copying here means setting up a new KD tree that has the same data but does not share any pointers. This is so that the first can be removed entirely and the second will still be valid. Data from the branches and terminals are copied into the KD tree and terminals are added to the names map and Group Map. With this base, then a loop is run:

```
for (unsigned n{ 1 }; n <= location.first ->v->n; ++n)
{
        addNTGroups(n);
};
```

This will add all the groups starting at 1 and going until all the terminals in the facade enter a single group.

Nodes need to be removed once they are no longer usable, and the central idea is to preserve order in the data structures. Repeated groups are stored in the SplitCost function to incur a cost for splitting them, but single groups are not because whether a line splits them or not, no repeat rule will be eliminated. If a split line leaves only one group of a certain type it needs to be removed from the splitCost structure for it no longer matters if it is split.

```
/*              NotLastAll   split there and more than one left
*               LastSplitOnly   Last means last Element Remove the split-- splits are there
*               LastSplitRemoved split already removed
*               LastAll         last and all needs to be removed
*
**********************************************************************
*********/
        enum  RemoveType { NotLastAll, LastSplitOnly, LastSplitRemoved, LastAll};
        GroupMap::const_iterator removeGroupPair(GroupMap::const_iterator it,
RemoveType );
```

removeGroupPair will remove a Group appropriately under different circumstances. If the group is repeated and we are not removing the last element the first option will remove the group from the SplitCost, Group Map and LL Corner. The second option removes it only from the SplitCost. This is appropriate if the group is single and did not repeat. If the split was already removed, the third option is used for it includes different error checking. The fourth also removes it from the name map.

`

There are three functions for removing Nodes that use *removeGroupPair*. *RemoveSingles* is run after all possible groups of a certain type are generated. In Group Map, say we have checked each group to the left of Node A and found only one A B group. At that point, there is no other way of forming A B, so that is a single; it is removed with LastSplitOnly, meaning it stays in the names map, the single is still in the Group Map and in the LL Corner structure but not in the splitCost structure. Next situation is when a split line is used, and those nodes that are no longer usable and need to be removed. That function is void *removeNodes*(NodeMap& nMap) where the NodeMap is the map of nodes split by the line. Depending on the number of nodes of a type, they may be singles or repeats, and in each case this function removes them appropriately. If no groups of this type are usable, they will be removed from all the data structures. The last structure is also *removeNode*, but this time removes a single node. That may leave behind a single Node of that type and so it also removes the splitCost in the other Node. In all cases after running the function the data structure keeps track of singles and repeats but only repeat groups are in SplitCosts and incur a cost of splitting.

5.0.7 Debugging

The NT Groups need to be stored in several places and it is easy with a wrong index or location to make a mistake. One error checking routine is this:

```
/**************************************************************************
************************************
 *  bool checkGroupPairStorage will check if a GroupPair is stored correctly
 *  @params[in]   thisNode  current Node,
 *                minVal ll corner,
 *                ntGroup   GroupPair of tested Group
 *                last      means the GroupPair is the last one of its kind. It was
 *                          inserted and the splits then removed.
 *                termFound  true if the term was found
 *
**************************************************************************
******************/
                 bool checkGroupPairStorage( std::shared_ptr<const Node>   otherNode, const
EVector& minVal,
                                    const GroupPair& ntGroup, bool last, bool& termFound);
```

For each Node added, checkGroupPairStorage starts looking at the rootNode of the KD tree (otherNode and MinVal) and checks whether ntGroup is stored correctly in each Node. It should be found in one and only one *LLCorner* (more than one causes invalid). *termFound* is true if one

`

terminal found in an LL corner. Every splitline for every node of the KD Tree is tested against the NT group without testing where the node should be. This does not make many assumptions about where overlaps should be found and is a good test.

Another similar routine is:

bool testAddingNodes(const BottomUp& bu, GroupPair pr, bool last);

and this one looks only where the ntGroup should be and it does not search the entire tree. It is faster but uses the same structures used when creating the tree, so less able to exhaustively find errors.

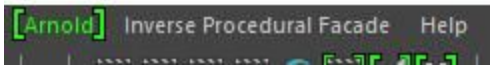
These both use a function *bool NodeSplit(SplitItPair split, const GroupPair& gpr, const GroupPair& ntPr, bool last) const*, that returns valid or not. *SplitItPair* chooses split lines within a node specified by *gpr,* where the groupPair tested is *pr*. If *pr* is stored in the split line it should overlap and if it is not it should not. If *last* is specified, then the the splitCost has already been removed and so should not be found at all.

5.1 The User Interface

Following describes the workflow of using this authoring tool.

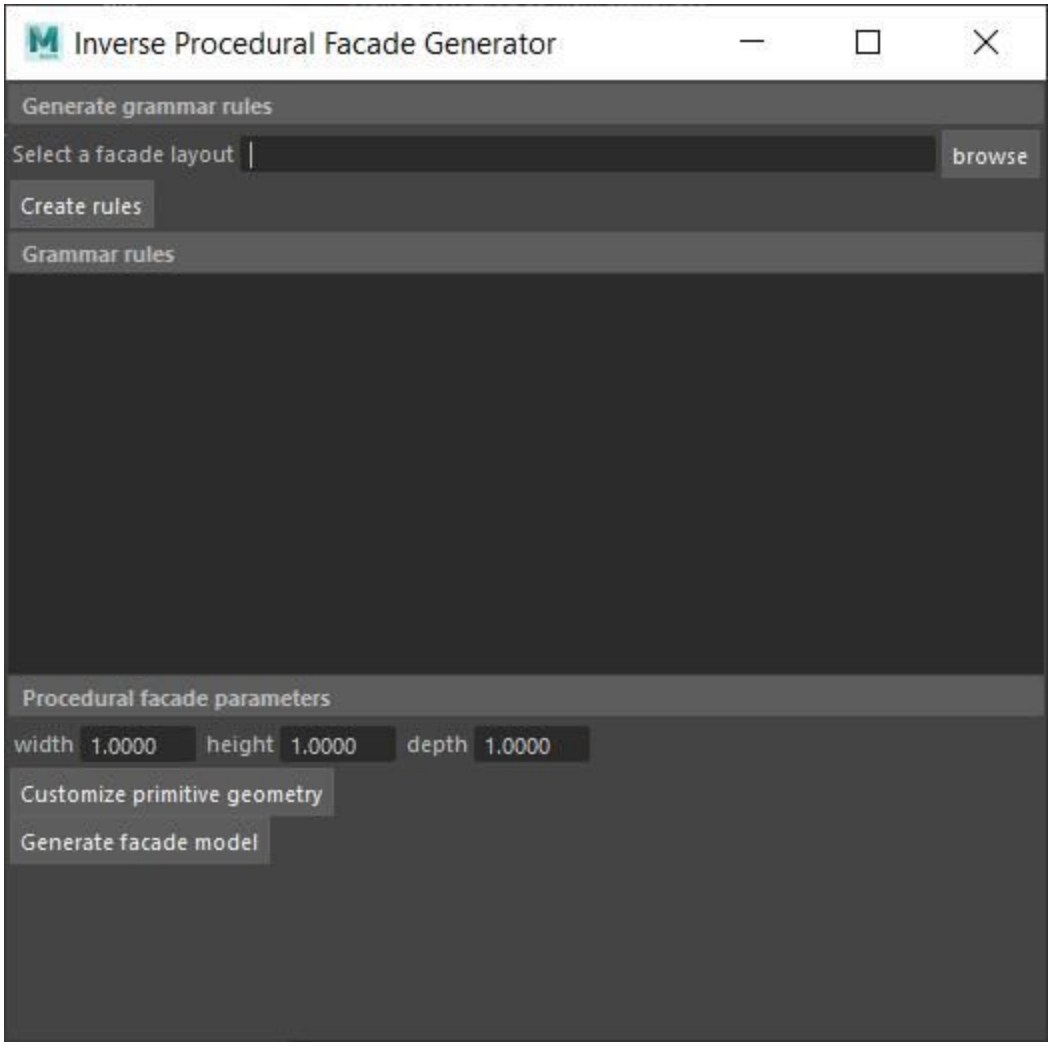
5.1.1 Open up editor window

After loading the plugin in Maya, there will be a menu item called **Inverse Procedural Facade** created in main menu bar.



Click on the menu and select **open plugin window,** and you will see the plugin window pop up as following:

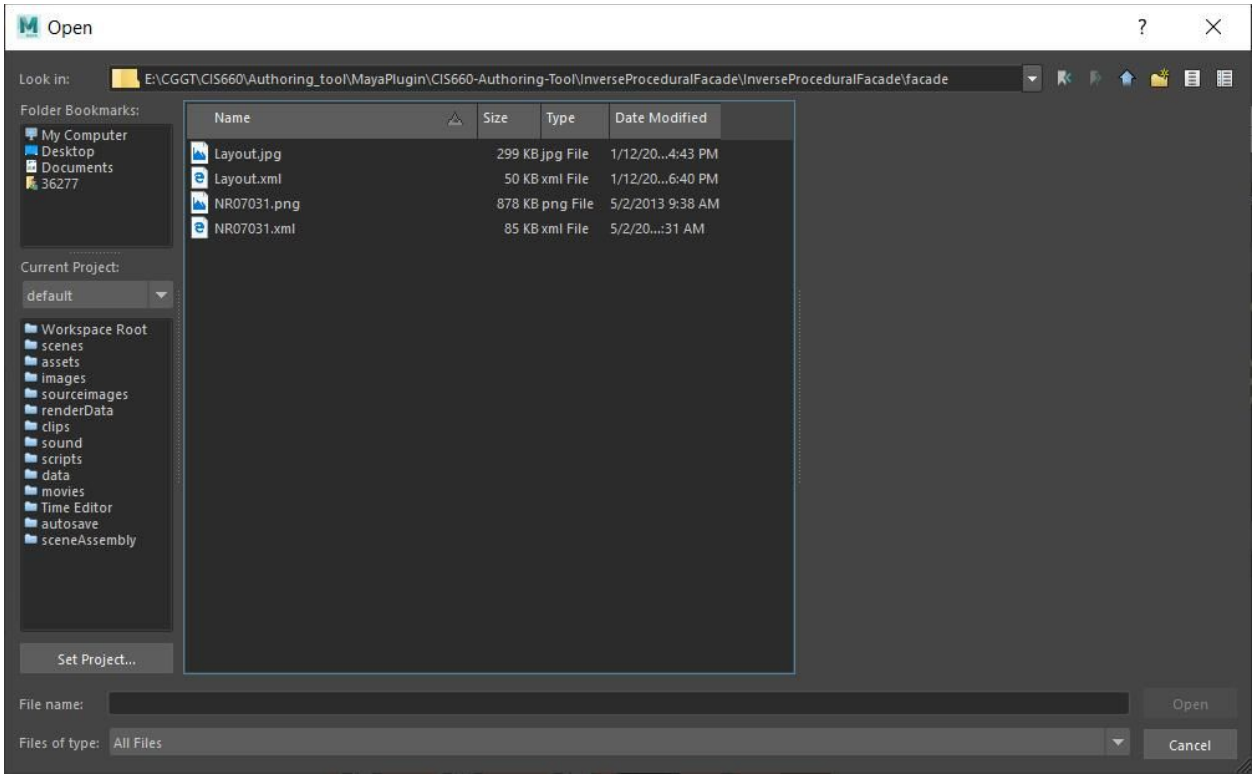
`



By clicking the **browse** button in **Generate grammar rules** section, it will show a file browser window where you can choose a .xml file which encodes facade layout.

`

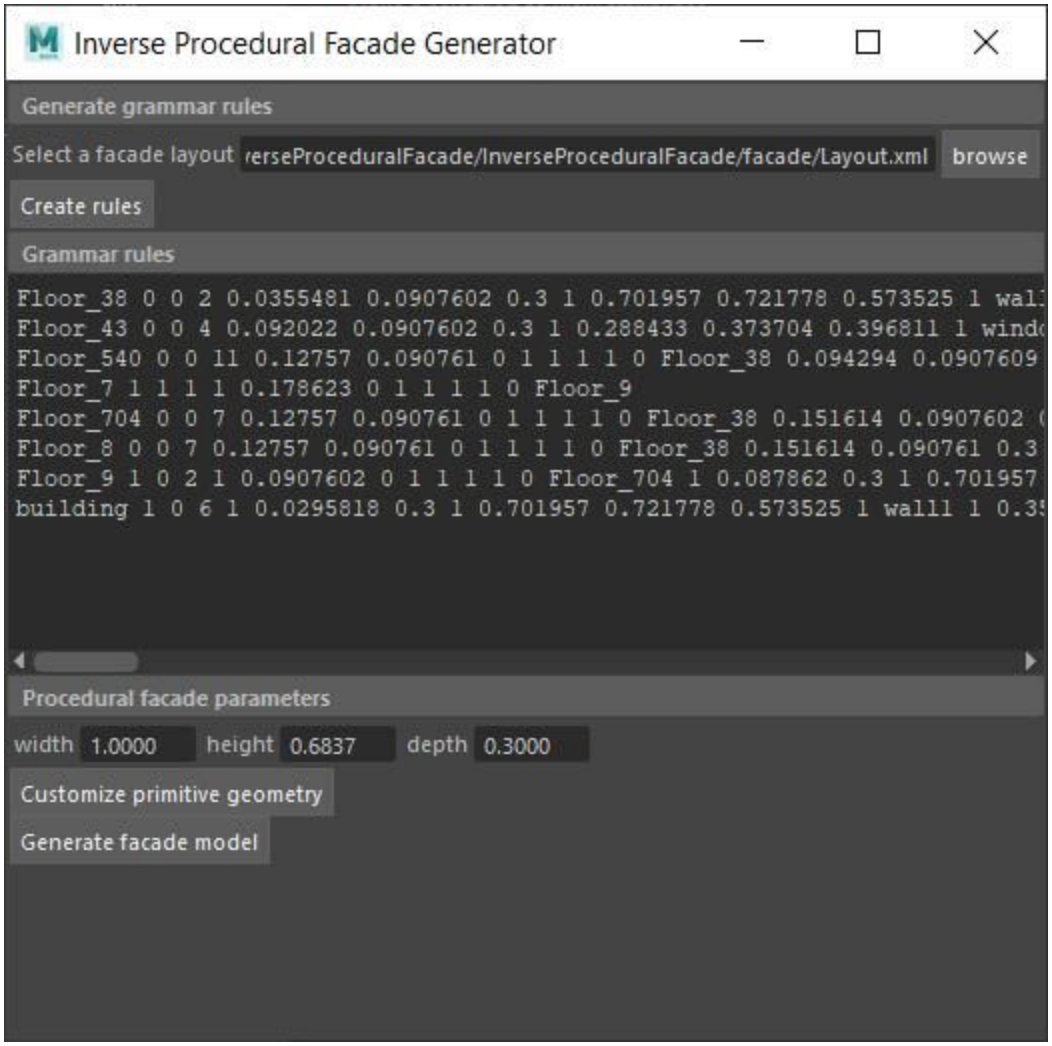
5.1.2 Browse input facade layout xml file



After selecting a facade layout xml file, click **Create rules** button below, the plugin will analyze the facade layout and run bottom-up and top-down algorithm and generate a set of grammar rules, which is then shown in the text field.

`

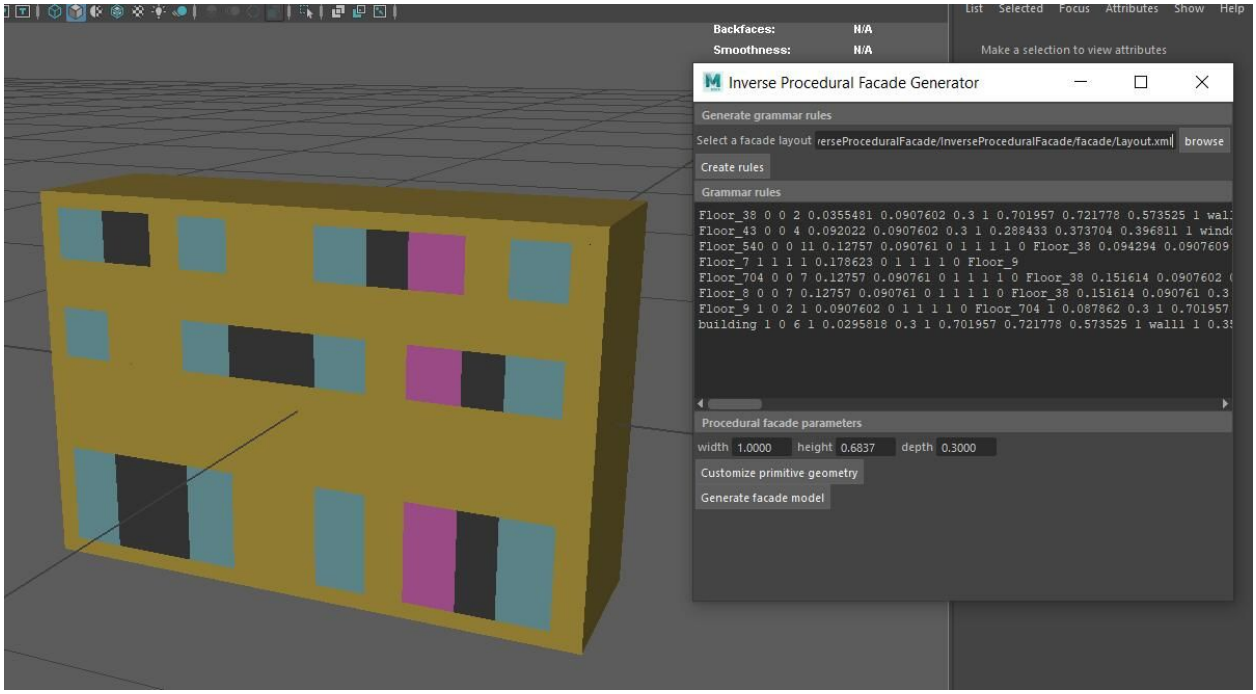
5.1.3 Generate grammar rules



5.1.4 Generate facade model

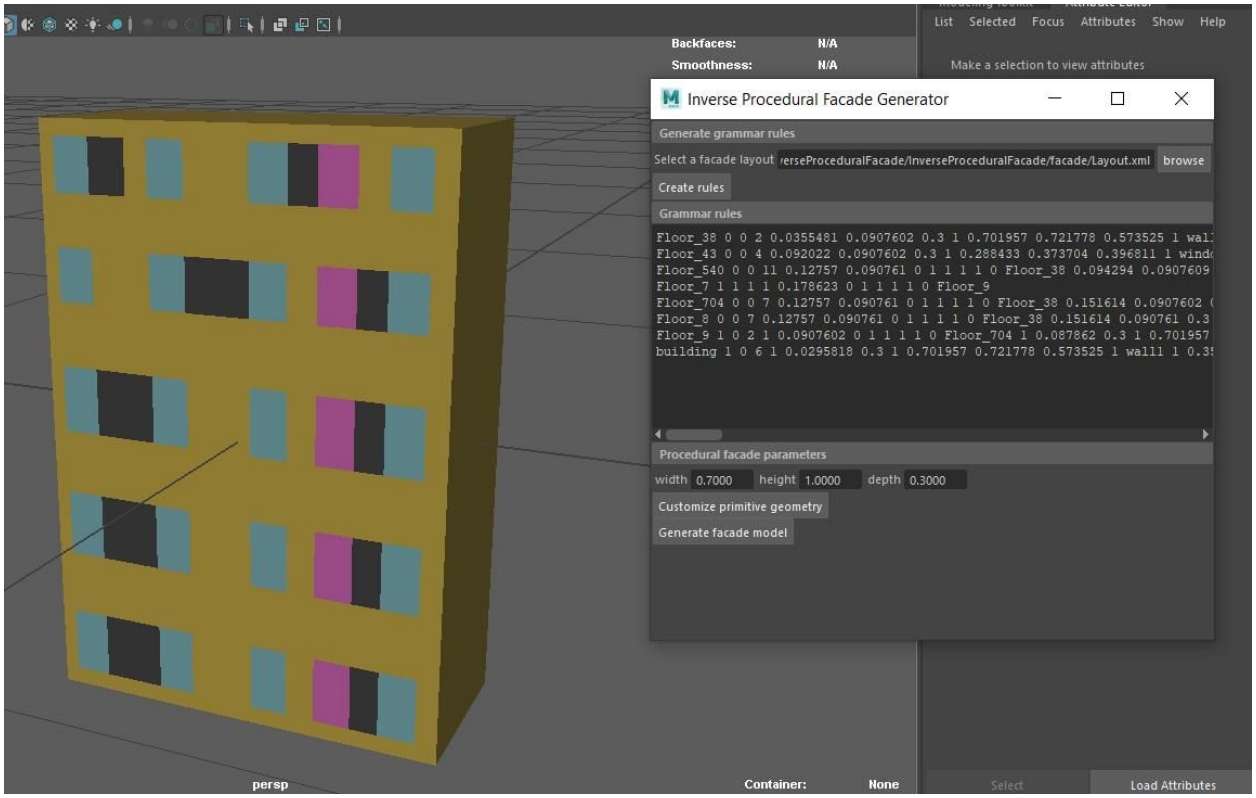
You can specify facade model size inside **Procedural facade parameters** section, otherwise the plugin would use default size for the model. Then, click **Generate facade model** button at the bottom of the editor window, you should see a 3D facade model is generated inside Maya viewport. Each color blocks represents the same type of primitive regions inside the facade layout, such as windows, walls, doors, etc.

`



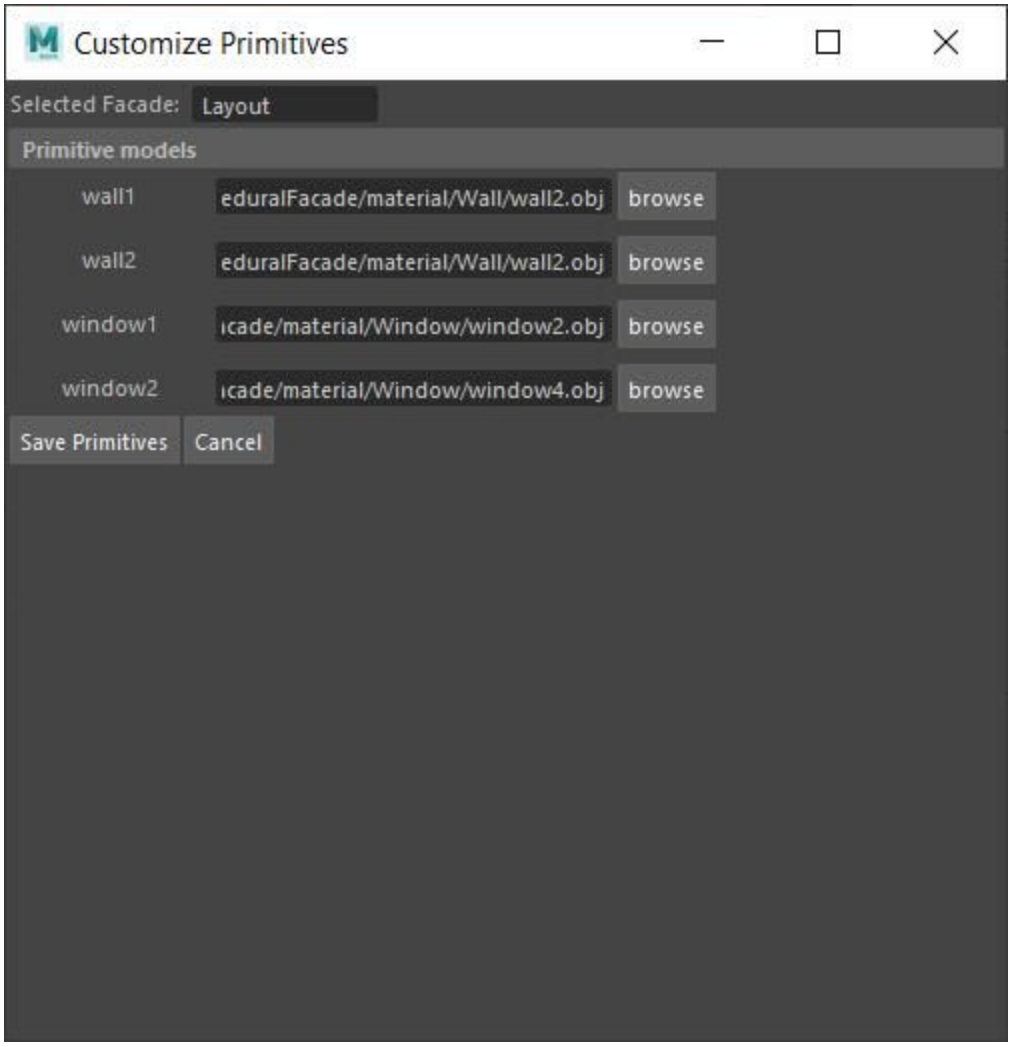
5.1.5. Resize Facade

Once the initial facade model is generated, you can resize the facade by changing the width, height, and depth parameters and the plugin would automatically update the facade layout while still maintaining the semantically meaning layout.



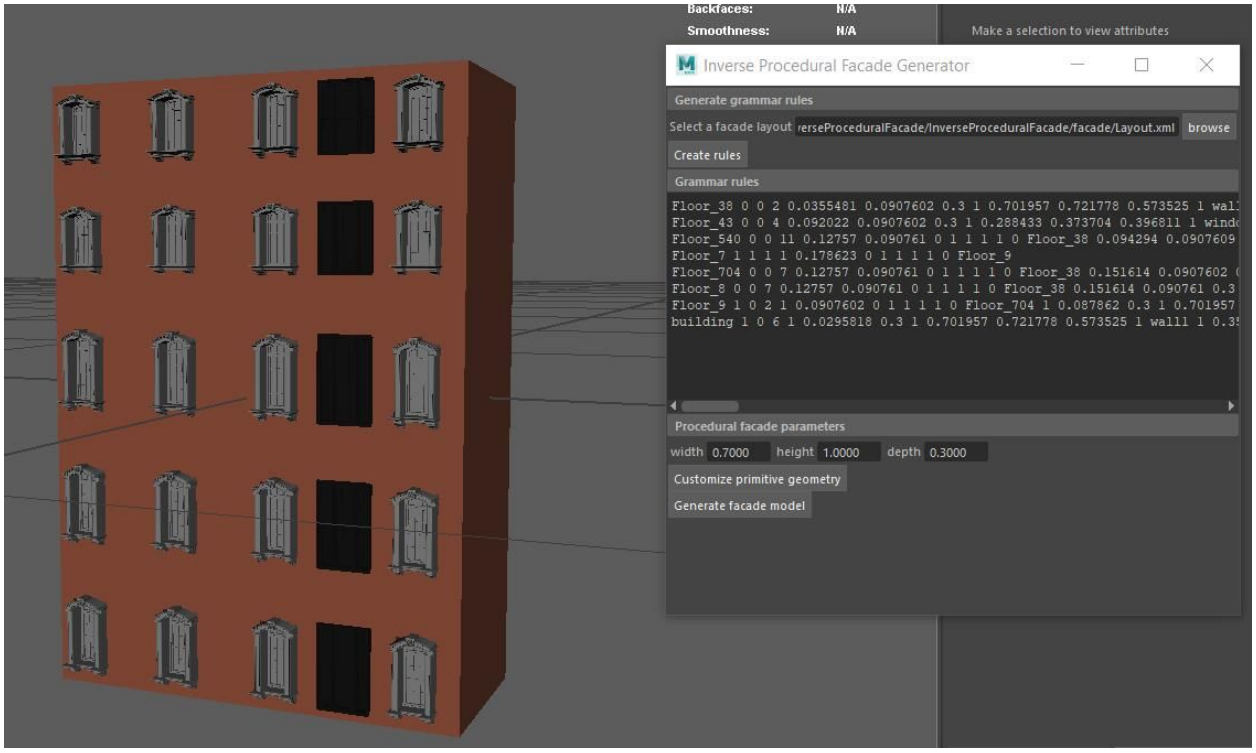
5.1.1 Customize Facade

You can customize the facade model by uploading your own primitive .obj files and assign them to each of the primitive regions. Click the **Customize primitive geometry** button and you will see new window pop up. For each primitive type, you can browse your own customized primitive .obj model by clicking **browse** button. Then click **Save Primitives** button to apply your changes to the facade model.



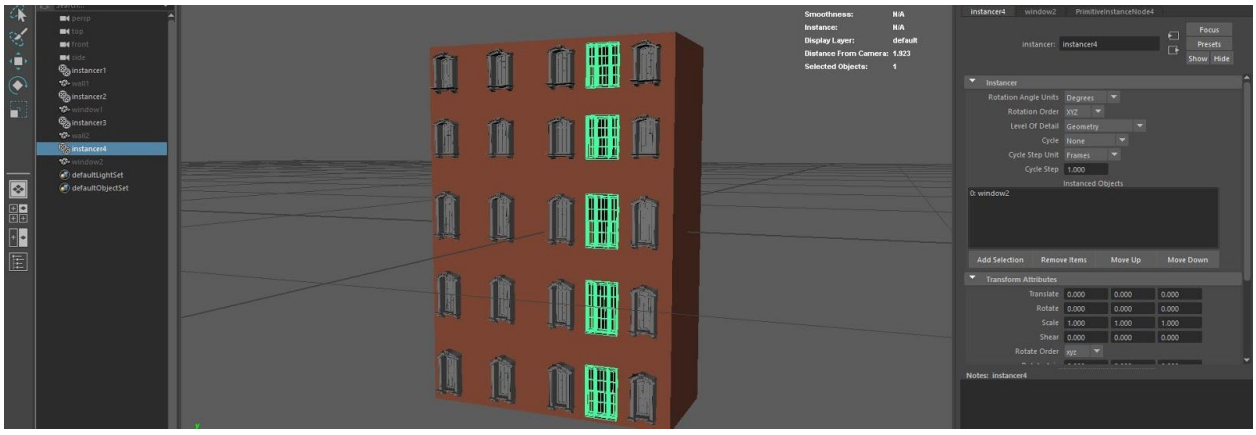
Now you should see that the primitive models are updated with customized ones:

`



5.1.2 Instance Node modification

In the outliner panel on the left hand side, you should see several instancer nodes and each of them is responsible for producing the same type of primitive models. You can click on any of them to see which primitive models are subject to. Once selected a instancer node, on the right hand side you should see a details panel of the selected instancer node. Here you could adjust transformations of all primitive models at once.



5.2. Content Creation

Following are example renderings produced by this authoring tool:

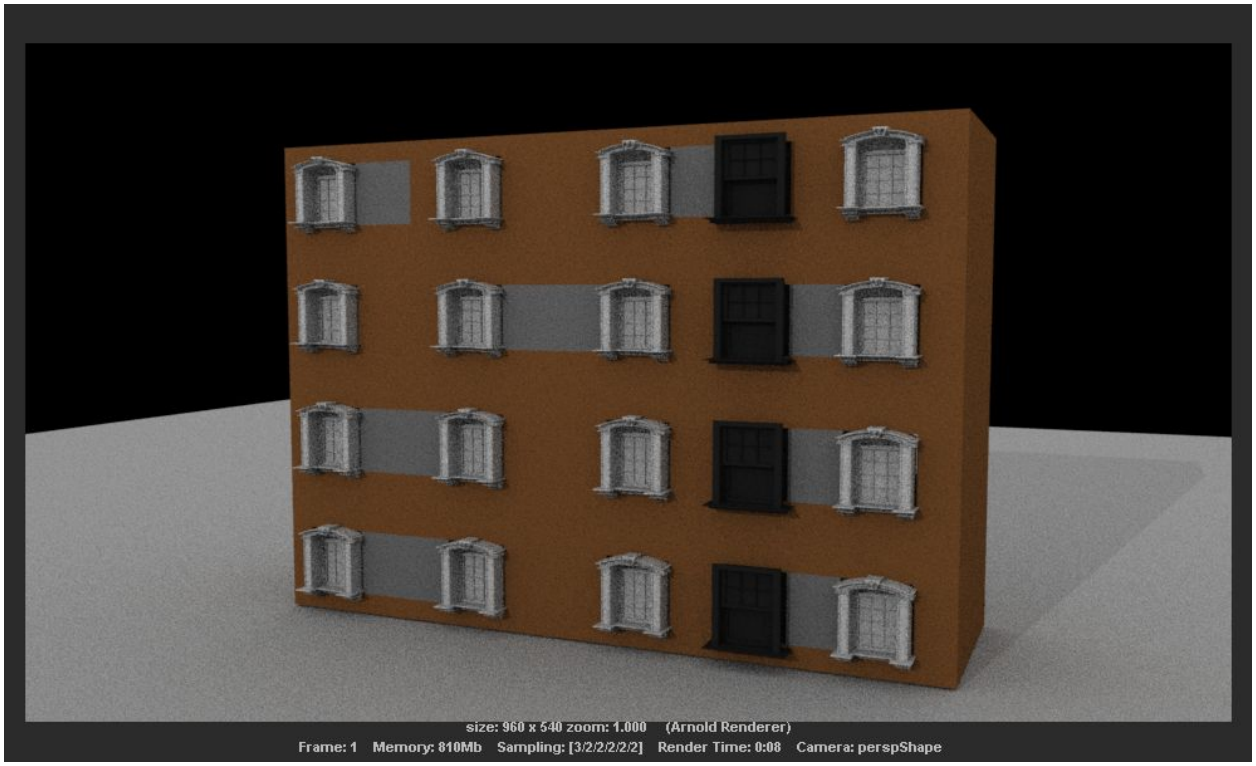
`

5.2.1 Layout.xml

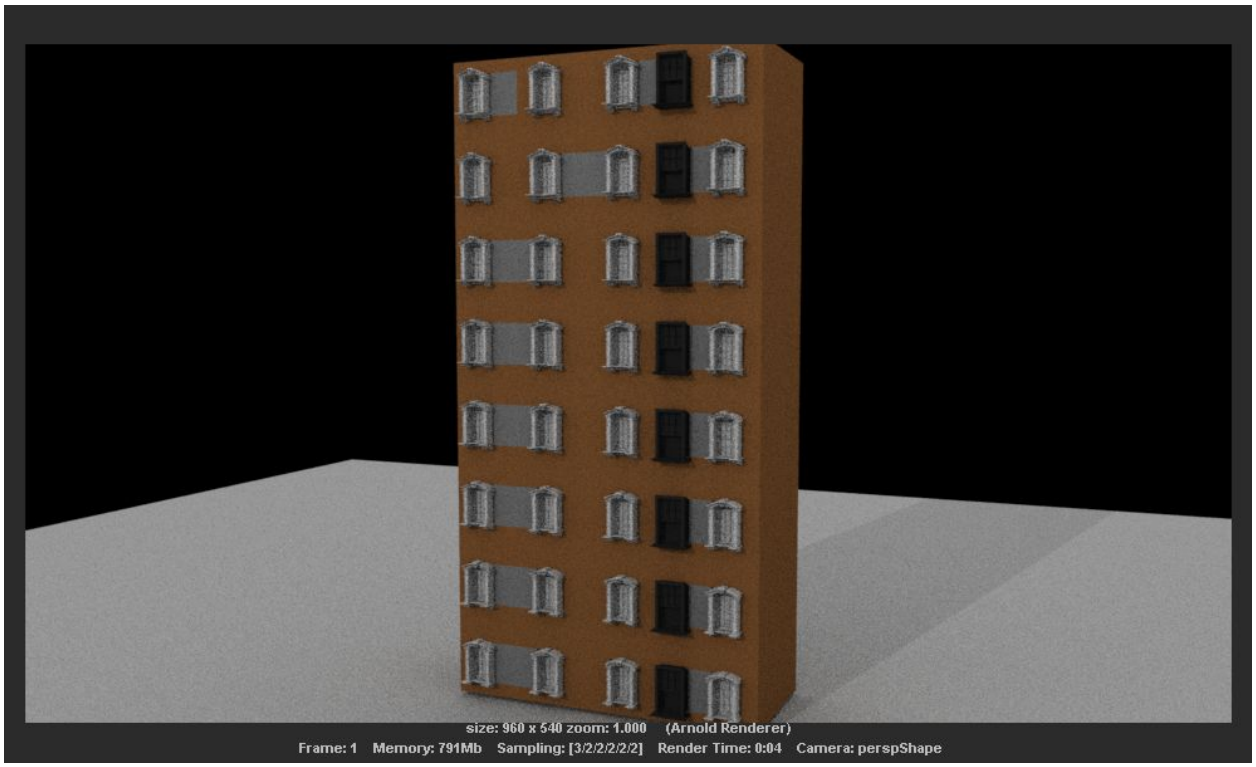
Original layout image:

Reproduced facade model:

`



Variations of facade model:



`

5.2.1 NR07031.xml

Original layout image:

Reproduced facade model:

`

Variations of facade model:

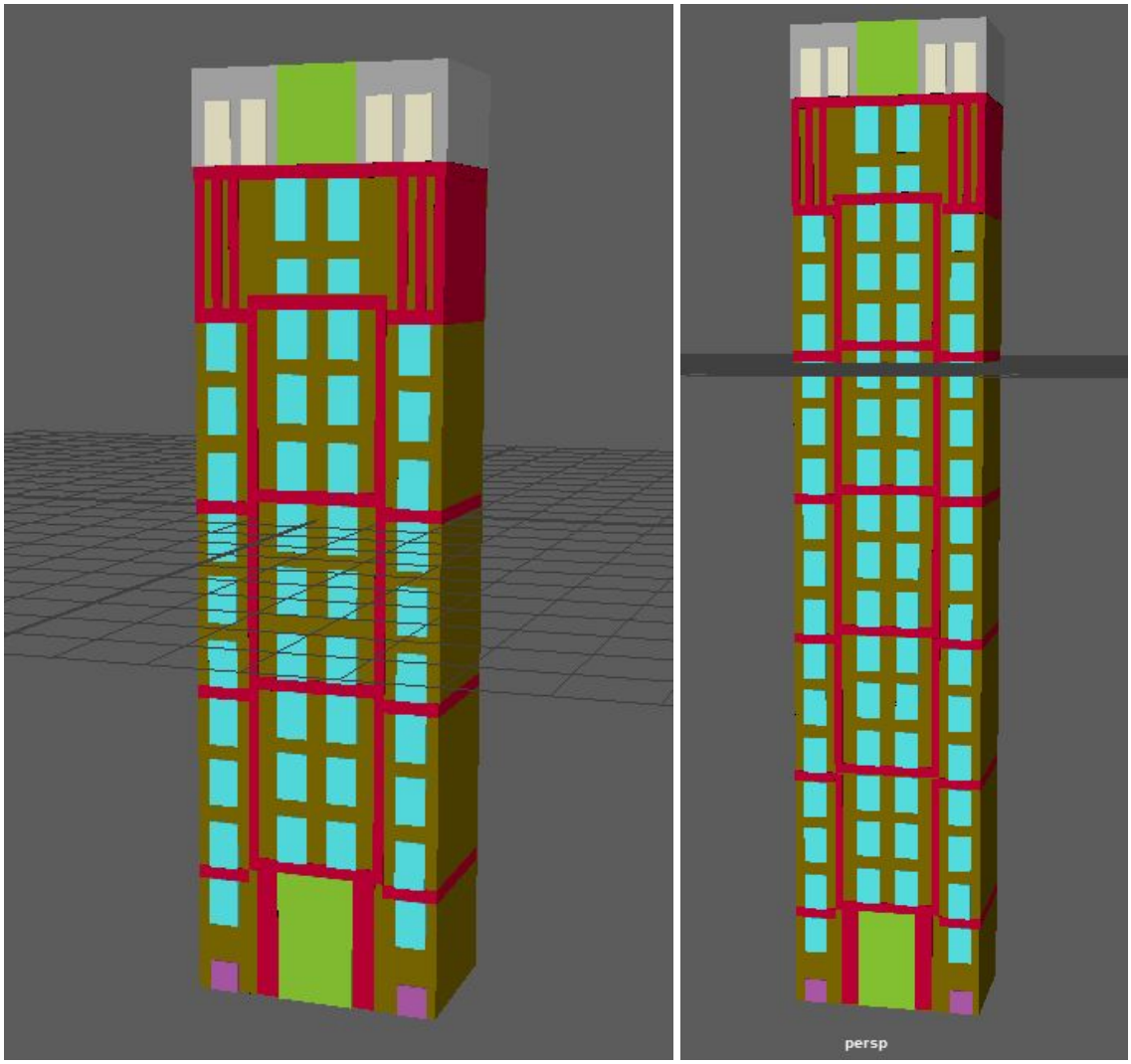
5.2.1 hight_rise.xml

Original layout image:

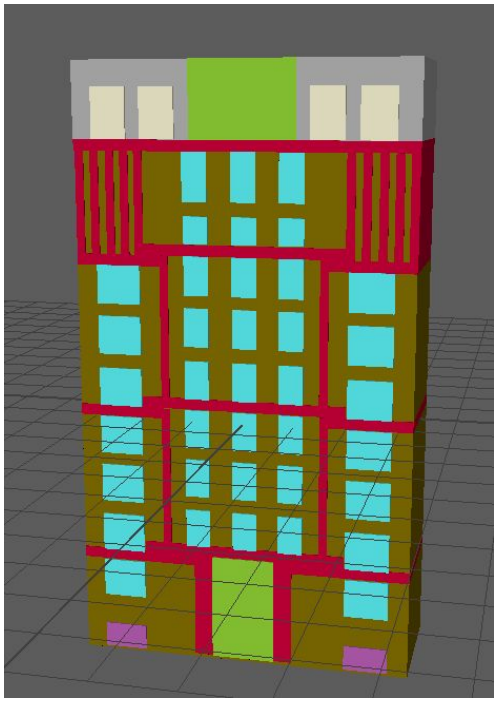
`

Variations of facade model:

`


persp

`



6. Discussion

6.1. Accomplishments

We have now a very solid algorithm for parsing the layout. It was critical to develop a solid way of creating all repeat groups, termed the bottom up approach.

All the data structures and functions, work and pass the debugging stage. Every group is checked against the entire tree to see whether it is found where it should be found and not found where it should not be found. In addition to the asserts that run these tests, every error condition is tested for and would generate a runtime error if found. None of those get triggered.

One example of a successful layout generation is high_rise.xml used above. High rise consists of 357 repeated terminals and 15 different terminal names. Once all the non-terminal groups are created, there are 229 unique groups, and 1533 groups found, on average 6.7 repeats per group type. As described under *Group Map*, the nonterminal groups are searched for sequentially, starting with nt groups with 1 terminal. After groups with half the number of terminals have been made, no more repeats are possible.

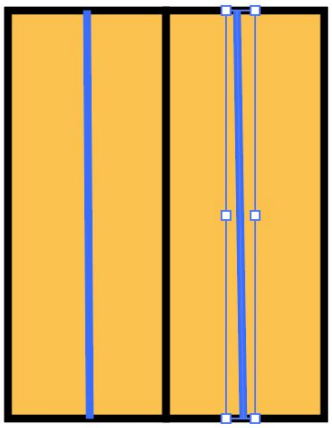
Single groups are kept track of and this is crucial for searching exhaustively for split rules. The figure in Section 5.0.5 illustrates the need. Once the algorithm arrives at the rectangle in the figure if we only had the KD structure we could only split the rectangle in either X or Y according to which way the KD tree was built. The best rule and lowest cost rule may *not* be along this axis and may be found along the other axis. With the KD tree alone getting rules for making a split along the non preferred direction would require navigating down the tree and

`

searching children for common split lines, which would be difficult. With the LL Corner structure all possible groups, even single groups, are stored and retrievable at each corner. regardless of the axis selected in the KD tree. One just needs to use *findXYLocMap* with the axis and the width of the rectangle and all possible NT groups will be found.

This works. For example in the high_rise xml, there is a single group with 357 terminals, meaning the entire facade. Running *findXYLocMap* with a fixed X width of the entire facade yields five nt Groups, including the largest group, corresponding to the 4 split lines in the first KD Node and the entire facade. In this case searching along Y == max size yielded only one Node (the entire facade). The LL Corner structure retrieves all possible groups without the limitations of the KD tree.

The next feature built that works is the Split Cost function and this simplifies choosing the best split lines. For example below is a rectangle with one group repeated twice illustrated in green. The entire rectangle could be split in 7 possible ways, (2^3 - 1), three of which are single line splits. One of them, that uses the black middle line, would preserve the two green groups and so would not incur a low cost because would not split the green groups. The two blue lines would incur a high cost each. An exponential function could be used to map high cost to low probability. With this cost function, high probability splits would be more likely.



6.2 The Top Down

The top down that we implemented used a greedy algorithm to parse the layout. It used the splits in the KD tree and looked for repeated adjacent groups. The two functions used were *findXYLocMap* to find all NT groups at this corner that split the entire rectangle, and *findLLNode* to to navigate to adjacent corners. We searched for patterns like A A A where the smallest group repeats, and patterns like A B A B or A B C A B C, and patterns like B A A A, where a later group repeats. The algorithm was greedy because if a repeat was found it was used; first non repeat regions were described with split rules and then the repeat region was described with a repeat rule. This generated a grammar set that captured many repeat structures and was straightforward to implement.

`

6.3. The Top Down: Next Version

The limitations of the greedy algorithm are that selecting repeats at a higher level may not yield the lowest cost rule set; it may make more sense to forgo the current split in favor of a better future one. The algorithm below is simple to implement and explores all possible rule sets looking for the best rule set. This problem has an exponential number of possible solutions so an approximate solution must be found.

Now with a cost structure solid and defined and with LL corner keeping track of singles, we are well positioned to write a better Top Down algorithm. The algorithm would start by making a copy of the BottomUp structure using the copy constructor (using *BottomUp(const BottomUp&)*. For each grammar rule set its final cost will be calculated by assigning a cost for every grammar rule and summing up costs until the facade is parsed. The lowest cost grammar rule set would be chosen. The paper describes a cost for using a split rule and for using a repeat rule in a rectangle. The paper also describes adding the cost of breaking potential groups as the the *SplitCost* structure implements; These rules would establish a cost for every rectangle split.

Each rectangular region also corresponds to one unique group in the Node Map structure. Once a rule has been established for that group, it is reused the next time that group gets divided. This minimizes the rule set.

Using the LL Corner structure , it may be possible to split a given rectangle along X or Y. If there is a choice one is chosen randomly (equal weighting). This allows for a much more thorough solution than if we only split along the KD tree.

To split a rectangular region, if there are repeats one option is to cut the region to preserve the repeats using the current greedy algorithm. Another is to randomly select a set of lines and use the cost functions to choose the most likely set. One of these two options would be chosen randomly. If the second were chosen, then the number of lines to use in the split rule would also be randomly chosen. This is important because using fewer split lines lowers tho cost now but increases the cost in later stages. For each set of split lines with the name number of lines, we would register a cost based on the split rule and the number of nt Groups rendered unusable. The actual set chosen would be chosen probabilistically weighting the lower cost sets more. Sets of higher numbers of splits are chosen equally often to sets of lower numbered splits.

Once a split set is chosen those groups that are no longer usable are removed from the data structures. The children are then processed for split rules. When a child Node is split it is also removed from the structures for it will not be used again. The remove function described earlier fullfill this role.

`

An optimization would be to calculate the rule for a particular group up to a fixed number of times and then select the best rule set for that group.

6.4. Total Man-Hours Worked

Yuru Wang:

- Research (10h)
- Meetings (20h)
- Coding (48h)
- Overall (80h)

Anton Khabbaz
research /reading 20 h, meetings 20 h, coding 100 h, overall 140 h

6.5. Third Party Software

We used tinyxml2, http://www.grinninglizard.com/tinyxml2/

`