TU WIEN Informatics

Prozedurales Modellieren unter Einsatz von Parser Generatoren

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Alexandra Gamsjäger
Matrikelnummer 01631843

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.techn. Bernhard Kerbl, BSc

Wien, 9. April 2022

[illegible]

Alexandra Gamsjäger	Michael Wimmer

Informatics

Procedural Modeling utilizing Parser Generators

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Alexandra Gamsjäger
Registration Number 01631843

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Univ.Ass. Dipl.-Ing. Dr.techn. Bernhard Kerbl, BSc

Vienna, 9th April, 2022

[illegible]

Alexandra Gamsjäger

Michael Wimmer

Erklärung zur Verfassung der Arbeit

Alexandra Gamsjäger

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 9. April 2022

[illegible]

Alexandra Gamsjäger

Kurzfassung

Das Modellieren und Simulieren von Gebäuden, Städten und Metropolen ist heutzutage von großem Interesse in den verschiedensten Industrien. Es erlaubt realistische Repräsentationen in einem dreidimensionalen virtuellen Raum zu beobachten, wie es auch in Computerspielen und Simulationen von spezifischen Situationen ist. Ein Weg dies zu erreichen ist die Prozedurale Modellierung, welche nur wenig Eingabeparameter benötigt, um komplexe Szenen und Strukturen zu generieren. In Kombination mit sogenannten Parser Generatoren kann ein effizientes und zeitsparendes Vorgehen erreicht werden. Die resultierende Anwendung ermöglicht es dem Benutzer schnell und simpel zu generieren, modellieren und designen von hochkomplexen Strukturen.
In dieser Bachelorarbeit werden wir zuerst den heutigen Standard besprechen und die Möglichkeiten, welche zurzeit vorhanden sind. Dann gehen wir weiter auf zwei bekannte Parser Generatoren ein, namens ANTLR und Bison und wie wir den erstgenannten mit dem zweiten ersetzt haben. Dies gibt uns einen guten Vergleich und Einblick in deren Vor- und Nachteile und soll die jeweiligen Endresultate demonstrieren.

Abstract

Modeling and simulating of buildings, cities, and metropolis is of huge interest in today's industries. It allows observing realistic representations in a three-dimensional virtual space as it is used in games and simulations of specific situations. One way to achieve this is procedural modeling, a method that only needs little input to generate complex scenes and structures. In combination with so-called parser generators, an efficient and less time-consuming approach is achieved. The resulting application then gives the user a simple and fast way to generate, model, and design highly complex structures.
In this thesis, we first will go into detail about today's standards and the possibilities which are available at the moment. Then resume with an illustration of two widely used parser generators called ANTLR and Bison and how we replaced the first with the second. Leaving us with a comparison, of their advantages and disadvantages and demonstrating the resulting outcome.

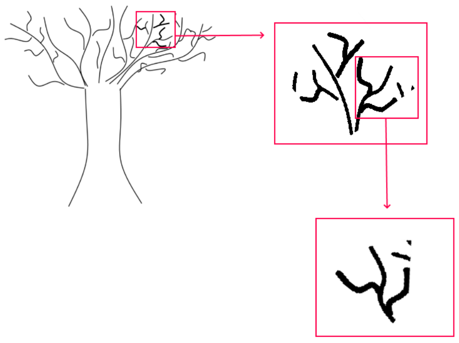
Contents

CHAPTER 1

Introduction

As Simulations, Games, and Visualizations get more and more attention nowadays and make up a huge portion of the entertainment industry, it is natural to want to perfect the development pipeline and shorten the time it takes to accomplish. 3D modeling is one part of this huge field of interest especially in content creation for games, which is getting very popular lately, but one of its biggest bottlenecks is the time it can consume and the tedious amount of work that occurs if large models are needed. Eliminating these bottlenecks would create more space to be productive and creative with the ideas and content someone wants to create. It is also the way to go, on behalf of trying out multiple different approaches in a less time-consuming way, being able to compare and evaluate these not only on paper, because before this would not have been efficient, as every little thing would have to be modeled by hand. That is why nowadays automatic model generation is a big topic trying to solve these problems, simplifying the process, shortening the work time and allowing having more time for important parts of the project. One big approach that arose due to this problem, was procedural modeling which solves the problem by letting the user specify attributes, limitations, and necessities via a grammar structure and then generating a model matching said rules automatically. There are many different examples where procedural modeling can be utilized to benefit the overall process. This includes the generation of self-similar objects, called fractals, such as trees, as branches themselves again look similar to the tree itself and the branches of them as well, and so on (see Figure 1.1 [a]).

Another great application is the generation of different vegetation-heavy scenes, including plants, plant growth, eco-systems, and even green-space planning in urban settings (see Figure 1.1 [b]). Because the modeling and simulation of plants get difficult fast, as they reach highly complex structures. This is especially of importance when incorporating the system of hormones being responsible for the plant's growth. One procedural technique we will touch on, which is capable of such a plant structure is L-Systems.


[a]

[b]

Figure 1.1: a) Self similar tree illustration. b) **Example Scene** Rendered plant ecosystem [DHL+98].

Figure 1.2: City Structure Example by Talton et al. [TLL+11]

Further, procedural modeling techniques also can be used to generate buildings, cities, and even metropolises, starting with simple structures and adding more and more detail to the buildings like facades, windows, balconies, and so on (see Figure 1.2). This is the main application we used in our bachelor project described in chapter 4.

Besides these examples, almost every structure can be generated by utilizing procedural modeling techniques, as long as the structure can be described by a set of rules. But the main applications, where procedural modeling is used, are complex, expensive, and large scenes that otherwise would be too cumbersome to create by hand. Another factor is, that some structures, for example, character designs nowadays, are most heavily dependent on artistic and recognizable details, which make them interesting and therefore would not be the best choice to use procedural modeling. On the other hand landscapes, vegetation, cities, etc. are often constructed of the same shapes, structures, and plants over and over, and are most often made to be one big background, for the recognizable characters to walk in.

In this bachelor thesis, we will explain the process of exchanging two different parser generators, which are tools to automatically generate code via a more simple grammar input and in our case further down the line also generate a 3D Object (.obj) model. Of course, it would be possible to write a handwritten parser to do the same work for us, but parser generators give us a tool to simplify this process and provide preset functions to end up with less handwritten code and consumed time in the end. The two generators mentioned are ANTLR [Par14a] and Bison [Pro14a], which are used to create such grammar rules for generating automatic 3D models of building, city, and forest structures. This includes describing both parser generators, discussing the differences, listing problems that may occur on either side, how exchanging these two was achieved, and what challenges had to be overcome. Further on I'm presenting an application for executing the result via a simple GUI to make the project more accessible for the user, without having to know too much about the program it inherits. This way the user only really has to know how attributes, limitations and necessities are written as a grammar to create a 3D model, immediately accessible to the user to utilize.

CHAPTER 2

Related Work

The following chapter will focus on research related to solutions to the problem of three-dimensional simulations and visualizations via procedural modeling. State-of-the-art approaches to procedural geometry, parsing, generators, and shape grammars will be discussed.

2.1 Procedural Geometry

As already mentioned in section 1 procedural modeling is a technique where models are generated automatically matching a set of predefined rules. The following subsection will explain multiple state-of-the-art approaches for this technique of creating models.

2.1.1 Model Synthesis

Model Synthesis is a simple procedural modeling technique, using simple 3D shapes as input for generating similar, but more complex models automatically. As well as allowing the user to set geometric constraints to control the outcome of the resulting model and preventing the algorithm to generate unnaturally large or small models, like Paul Merrell and Dinesh Manocha have done in [MM11].

2.1.2 Declarative Approach

One way to simplify procedural modeling for a user is a declarative approach, which provides different tools, so the user does not have to think about how the world is generated, but more so focus on knowing what should be created. Two of such approaches are mentioned in [STdB11] by Smelik et al. namely interactive procedural sketching and virtual world consistency maintenance. Interactive procedural Sketching provides, similar to other visual editing applications, different tools for the user to sketch the rough map of a virtual world in 2D. This is realized within two interaction modes, the Landscape

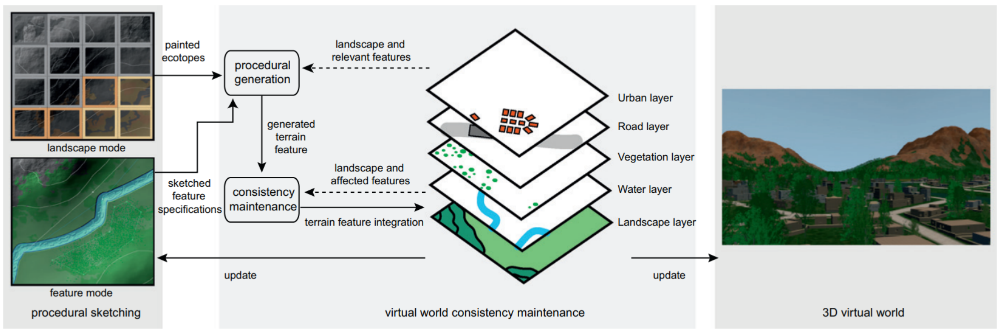


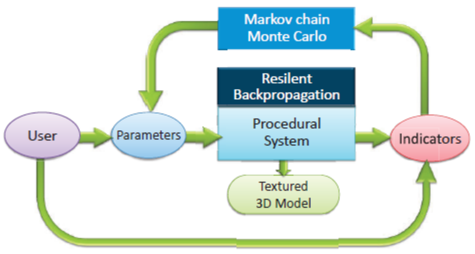
Figure 2.1: Workflow of the declarative modeling approach. Split into procedural sketching and virtual world consistency maintenance. [STdB11].

Mode, which includes defining elevation and soil material of the landscape, painted with so-called ecotopes, and the Feature Mode, providing several features like forests, lakes, rivers, etc., which are then placed via vector lines and polygon tools. All elements are then procedurally generated within the preassigned sketched areas. The virtual world consistency maintenance on the other hand takes care of the resulting dependencies between the designed features after sketching the map. That is because if the virtual world should resemble the real-world circumstances, these features would affect not only the terrain but also other features nearby. See one example of the workflow in Figure 2.1.

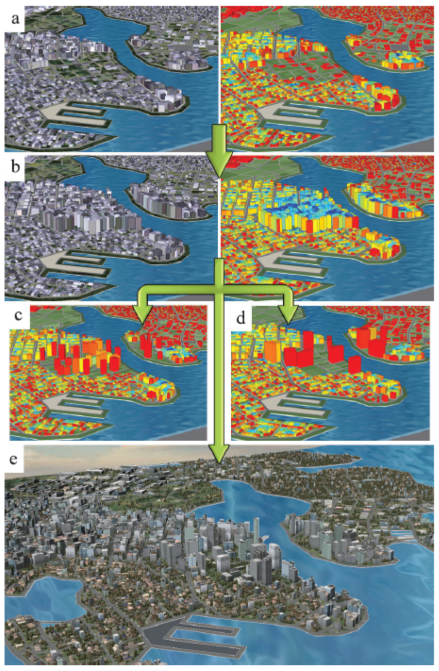
2.1.3 Inverse Procedural Modeling

In inverse procedural modeling, the approach is to take an already existing model and from this point of origin procedurally generate/model one or more new models. These new models mostly then, in some kind, resemble the original model with a few alterations made. This is a huge topic as it creates the possibility to generate multiple different outcomes from only one initial already existing model. Reducing the input needed to get a variety of models similar in some aspects but diverse enough to see a difference. This is a good opportunity to use, on designing entire cities worth of models, by only altering one initial model multiple times. This also reduces the time the user has to put into formulating different procedural rules-sets to achieve a variety of models.

One example to use this approach is to control an existing model and interactively edit it, without the user knowing the process behind the application. Vanegas et al. did this [VGDA+12] with an inverse urban modeling algorithm additional to the usual forward procedural modeling process, to optimize input parameters. This allows changing a model, without having to re-write the entire procedural rules. These changes get triggered by the user specifying local or global target indicators. These indicators can, but do not have to have an obvious relationship to the input parameters and can vary from very simple altering descriptions of parameters to complex semantic metrics. These include sun exposure, floor-area ratio, interior natural light, distance to the closest park, and more. This allows a high level of abstraction from the inherited process and manipulation



[a]



[b]

Figure 2.2: a) Coupled forward inverse modeling pipeline. b) Local Changes for Global Indicator Control [VGDA+12].

of indicator targets in an interactive way. With this approach then various, different solutions that match the set values are found, which is achieved by imitating indicator behavior through back-propagation and then letting the Monte Carlo Markov Chains (MCMC) engine evaluate the indicators for a large number of iterations and instances of urban models and choosing the best solution states of all (see Figure 2.2 a for system pipeline). This means the MCMC engine is exploring more models than the end-user is seeing, as only the best solutions are presented in the end, determined by filtering solutions, where the fit for the estimated indicator values and the actual values is low. Because this procedure inherits multiple iterations, which can get very expensive for every little model change made, the procedural and indicator measurement system got replaced with a neuronal network, by Venegas et al. [VGDA+12, p. 5]. This is, because a neuronal network as it is, can quickly estimate indicators after it got properly trained with valuable data. In the end, the user only has to use a slider to alter parameter values in the forward modeling aspect or set the indicator values through the inverse modeling, which showed great results and were completed in under 5 minutes. One example of a result can be seen in Figure 2.2 b.

Another great way to use inverse procedural modeling is presented by Bokeloh et al. in [BWS10], which uses symmetric features of the input model and further on generates a new exemplar by creating local neighborhoods which match local neighborhoods in the input model. This is then achieved by extracting rules matching the input model, which

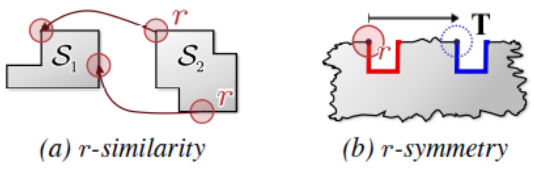


Figure 2.3: (a) r-similarity: within a radius of r every point in S2 is locally similar to a point in S1. (b) r-symmetry: within a radius r, symmetric points have to be similar under a fixed transformation T [BWS10].

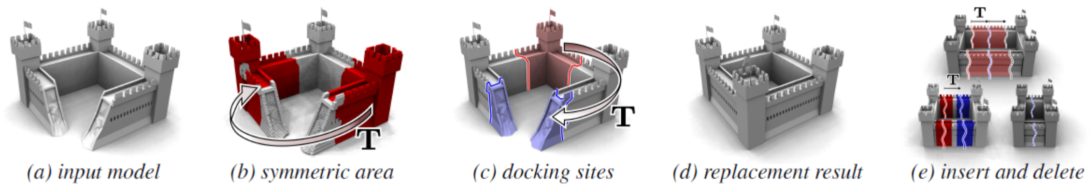


Figure 2.4: (a) Input model. (b) Symmetric areas marked in red. (c) Docking Sites: curve running through symmetric areas cutting out docker regions and partitioning the model into different parts. (d) Replacing a docking site with the help of a shape operation. (e) Insert and delete operations [BWS10].

describes the structure and can be then used to semi-automatically model shapes similar to the input. To describe and generate new models procedural rules are automatically constructed, which then is the so-called *inverse procedural modeling.* The similarity is described as, any point of the generated model that matches a point on the input model within a local neighborhood of set radius r, resulting in a strict r-similarity for every newly generated model, which this paper does guarantee for their approach.

As mentioned before symmetric features of the input are used to describe the model, this is defined as followed: "Two points of S are symmetric under a transformation T if T maps between them and their infinitesimal local neighborhood are topologically equivalent." [BWS10, p. 3]. Further on they also describe that a point is then r-symmetric if its whole r-neighborhood is symmetric under the same transformation [BWS10, p. 3]. This is then used to describe a simple algorithm for calculating a symmetric area. For a better understanding see Figure 2.3. But as usual, not every object is always perfectly symmetrical, therefore Bokeloh et al. use the term *dockers* for every non-symmetrical region and *docking site* for a curve running through symmetric areas cutting out the dockers and partitioning the model into different pieces. This makes it easy to exchange these docker regions with new geometry or no geometry at all by using different shape operations, while simultaneously maintaining similarity. See Figure 2.4 for a visual explanation of the different definitions.

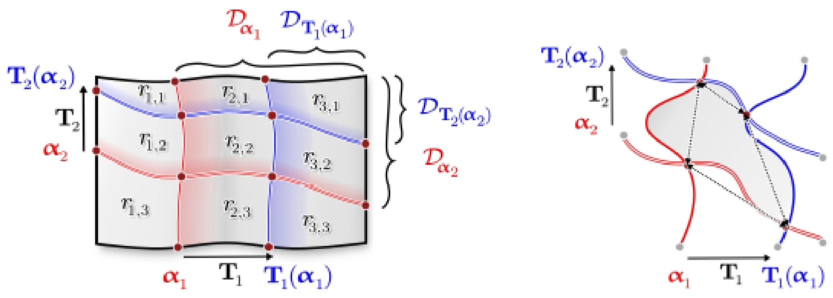

$\mathcal{D}_{\alpha_1}$
$\mathcal{D}_{\mathbf{T}_1(\alpha_1)}$
$\mathbf{T}_2(\alpha_2)$
$\mathbf{T}_2$
α_2
$r_{1,1}$ $r_{2,1}$ $r_{3,1}$
$r_{1,2}$ $r_{2,2}$ $r_{3,2}$
$r_{1,3}$ $r_{2,3}$ $r_{3,3}$
$\mathcal{D}_{\mathbf{T}_2(\alpha_2)}$
$\mathcal{D}_{\alpha_2}$
α_1 $\mathbf{T}_1$ $\mathbf{T}_1(\alpha_1)$
$\mathbf{T}_2(\alpha_2)$
$\mathbf{T}_2$
α_2
α_1 $\mathbf{T}_1$ $\mathbf{T}_1(\alpha_1)$

Figure 2.5: (a) Colliding insert/delete operations resulting in regions of different types. (b) A not tileable grid cell resulting from an arrangement of docking sites [BWS10].

The description for generating such similar models is encoded inside of a shape grammar, where they used three variants to accomplish this, general rewriting systems, which then get computed into a context-free subset of this grammar, and then supplemental grid structured production rules are added. This means they take advantage of the general rewriting system, as it is very expressive, but use context-free grammar to make it more computationally accessible, and at last, they add these grid-based replication rules for analyzing real-world objects [BWS10, p. 5]. Grid-based replication rules are built on the fact, that real-world objects, especially man-made ones, contain or are composed of grid structures. An example would be windows in a facade of a building being of any dimension *n1 x n2*. Because context-free grammar can not represent grids of two or more degrees of freedom (DoF), these grid replication rules are added. Delete and insert operations on the docking sites are defined as one DoF. But if a collision of multiple DoF docking sites is present it is a Dof of two and can't be handled by a context-free grammar. The grid replication rules achieve a classification of the different sections in a grid, deriving the right shape operation for each (see Figure 2.5).

For generating a basic shape matching grammar, matching docking sites are searched to then construct a set of shape operations, which can be applied on said docking sites, generating an r-similar model. This is a very expensive computation and is not compatible with standard procedural modeling tools, which leads us to the next step of extracting manageable subsets. In said context-free grammar they use non-terminals to define space where new geometry can be added on, being the docking sites, and terminals to then encode this geometry, being dockers. As usual, these non-terminals and terminals are used inside of production rules and in this case to describe a hierarchical structure of insertions of pieces [BWS10, p. 5]. Further on the context-free grammar is constructed by building a tree describing hierarchical dependencies of the docking sites and then going through this tree bottom-up to specify production rules for each node, where leaves are equal to a terminal. At last, a separate grid replication rule is added to describe the number of discrete degrees of freedom the model holds [BWS10, p. 6].

This approach can be used to create random shape variations and for semi-automatic modeling, which both are very useful in modern 3D modeling and help to simplify complicated processes [BWS10, p. 7-8]. Such a method gets relevant in cases of modeling and generating large areas and cities, as similarity comes in very convenient in these particular cases. This is because cities and metropolises are by their nature self similar, as buildings are often of the same structures (e.g. box with rectangles as windows, differ in dimensions, colors, etc.), with just some small differences in style. This of course is associated with the usual housings, not meaning special cases like for example, castles.

2.1.4 Statistical and Fuzzy Interference

Rather than generating 3D building models from scratch, using individual user-specified input parameters, another approach is using an existing architectural building to generate models with similar characteristics, being able to reproduce buildings of a certain appearance, which creates an entire model looking as if it existed in the same geographical location/scene and time as the original input architecture. This also gets interesting with the ability to simulate actual architectural heritage appearances. This can be achieved for example via CGA Shape Grammar (see further explanation in chapter section 2.2), combined with fuzzy and statistical analysis of the real-world building characteristics, as it is done in [TS13] by Tepavčević and Stojaković. Whereas fuzzy and statistical analysis is used to define these characteristics, procedural modeling is creating different variations according to these probabilities, generated by the fuzzy and statistical analysis. To be more specific, the statistical analysis compares characteristics of buildings and their parts, based on this the given buildings were segregated into groups by similarity. Further fuzzy logic is used to create a mathematical model based on the fuzzy set generated before. This model then gets described by a CGA Shape Grammar and results in a generation of virtual 3D models, similar to the given input buildings.

2.2 Formal Grammars in Conjunction with Shape Grammars

One of the first introduced approaches of procedural modeling was the so-called L-Systems, introduced in 1968 by Astrid Lindenmayer [Lin68] on behalf of discussing, computing, and comparing intercellular relationships of plants. This method got interpreted many times leading to different tools to model all kinds of structures nowadays [PL90, p. 3]. L-Systems propose a rewriting system, which is a technique used to define complex structures and objects. This is done by successively replacing parts of a simpler initial object, using a set of rewriting rules, also called productions. In the case of L-Systems, these productions are applied parallel and operate on character strings, which can further define commands leading to the creation of a structure.
Such a character string-based rewriting system denotes an alphabet, which is then used in the production rules to form the syntax of a valid string. Valid strings are defined differently depending on the purpose which is required. A predecessor, mostly written on the left side of the production rules, defines what is being replaced and the successor,

mostly on the right side of the production rule, defines with what this predecessor is replaced. Successors can further on be defined as a predecessor in another production rule, resulting in a repeated replacement of the starting symbol, called an axiom.

One interpretation of L-Systems is called CGA shape, which is "a novel shape grammar for the procedural modeling of CG architecture" [MWH$^+$06] and is used in the approaches in chapter subsection 2.1.4 and the next chapter section 2.3. CGA shape generates complex architectural models at a low cost. This is done with the use of context-sensitive shape rules, which define interactions between entities of the hierarchical shape descriptions. Production rules first generate a rough model of a building and iteratively evolves the design, by creating more details.

2.3 Building Worlds

From small units to big complexes, procedural models are used for creating models of different sizes. The following subsection explains state-of-the-art approaches for modeling buildings as well as big cities and metropolises.

Modeling Buildings

One way to generate extensive, low-cost architectural models is using CGA shape introduced by Wonka et al. in [WWSR03] and refined by Müller et al. in [MWH$^+$06]. CGA shape iteratively produces a model, increasing the details with every iteration using production rules, which firstly results in a coarse model of a building, then giving the model a more defined façade and in the end adding features like windows, doors, and ornaments. This procedure results in an embedded creation of a hierarchical structure inside of the modeling process, which is important for reusing design rules and creating complex city variations further on. This approach ideally combines an approach by Parish and Müller [PM01], which will be discussed more in detail further on, and an approach by Wonka et al. in [WWSR03], by altering both to fit and combine smoothly. This is important as Parish and Müller use simple models/boxes to represent buildings and further use shading to add details, which does not generate a sufficient geometric detail and comes with problems at intersections of different architectural elements. Furthermore, Wonka et al. only work effectively on simple mass models and changes increase the complexity and number of production rules by plenty. Müller et al. refined CGA shape by adding the repeat split, scaling of rules, and component split to the already existing basic split by Wonka et al.Ȧ basic split is splitting the current scope, e.g. the façade, along one axis, analog to the basic split, the repeat split tiles an element along an axis multiple times as there is space in the scope, this allows splitting on larger scales. For shapes, less than 3 dimensional, the component split is introduced [MWH$^+$06, p. 3]. Adding, scaling, translating, and rotating shapes are done by grammar notation and general rules, which are inspired by L-Systems [MWH$^+$06, p. 2]. With these operations, it then is possible to generate complex and detailed models, which first start to be simple primitives and

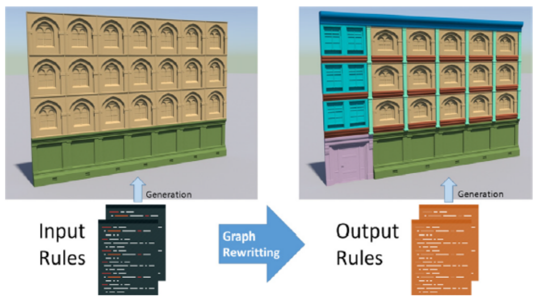


Figure 2.6: Applying graph rewriting techniques on a input model, resulting in a new model with a different facade [MP19].

then through further operations get more and more complex. In addition occlusions of elements and snapping of existing shape rules to a dominant face or line are implemented to create a more consistent design [MWH$^+$06, p. 4-5].

A recently introduced possibility for modeling buildings is a graph-rewriting /rule-set-rewriting approach which improves one of the biggest downsides of procedural modeling. This is the need to change already existing building models by a large amount and having to write an entirely new rule-set from scratch. The approach was introduced by Martin and Patow [MP19] and solves this problem and provides an initial rule-set for generating a completely new model. This should allow the user to use complex editing operations without further user intervention, apart from the initial creation [MP19]. This is accomplished by a graph-rewriting technique that is using graph transformations to modify the input rule-set to create an entirely different building. Starting from an initial production rule-set the individual rules can be rewritten by replacing the terms of a rule with another term, defining different transformations than in the initial rule. By applying these new transformations to all instances of the initial term the outcome results in a partly different model, as you can see in Figure 2.6.

This approach is based on graph-grammars, which means the graph's nodes represent the operations/transformations which are performed on the input geometry. The edges connecting the nodes, in turn, represents the incoming stream of geometry. With this, you can envision the formerly explained ruleset-rewriting as substituting parts of a graph (also called subgraphs) with a different part. The rewriting instructions by now consist of parameter change, node creation, deletion, and node reconnection operations. With these tools, the user then can create multiple new models by rewriting an initial model, by not only applying one new rule rewriting instruction but by using multiple new rules replacing parts of the initial ruleset in different sequences, which you can see in Figure 2.7.

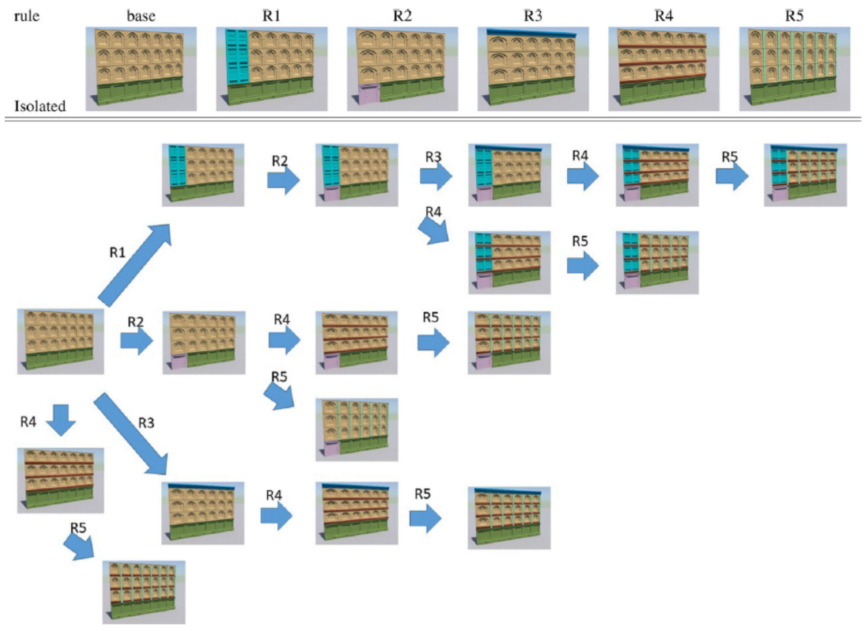


Figure 2.7: Applying different rules on a inital model in various sequences. The upper row shows each individual rule being applied on to the initial ruleset on its own [MP19].

2.3.1 Modeling Cities and Metropolis

Since procedural modeling is very well-fitting for the otherwise time and resource-consuming task of modeling cities there are multiple approaches to solving this task.

Different from generating a building or complex, the generation of cities inherits infrastructure, terrain, and much more, which makes it all the more complicated. One approach to accomplish this task is to use L-systems (see Chapter section 2.2) to generate a road map, including highways, streets, lots, and geometry for buildings, which is described in [PM01] by Parish and Müller. To generate such a virtual model, an input of multiple image maps is used, which represents populational, environmental, and other influences that shape the way cities are composed (see Figure 2.8). The specific system introduced in [PM01] is called CityEngine, which produces an entire virtual city from scratch using only little input. This is made possible by a hierarchical set of comprehensible rules, which still can be easily controlled by the user [PM01, p. 301]. Parish et al. concentrate on generating a traffic network and buildings, as they do not change very fast in an urban setting. For generating large-scale road patterns, the L-systems model is used and extended for the specific use of CityEngine, which makes it possible to set global goals, as well as local constraints. In a very simplified way, the system architecture then goes as follows, with the use of the extended L-System a Roadmap is created, a division into lots is done, and then the buildings also get generated by using the L-System. Everything after this is just the usual procedure of rendering and displaying the outcome to the user,

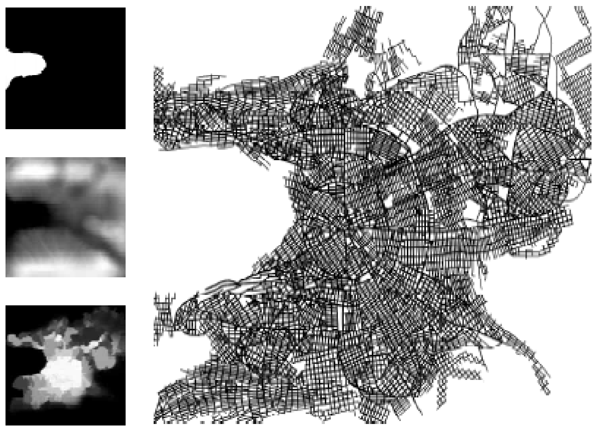
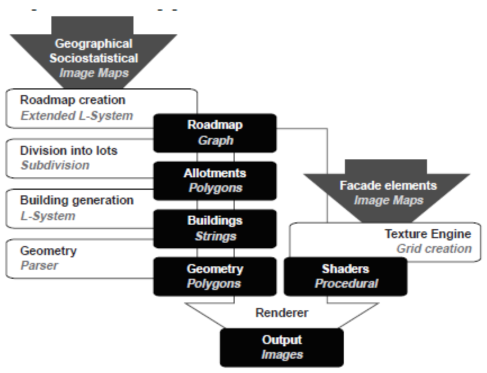


Figure 2.8: Left: Maps of water, elevation and population density. Right: A possible output generated by the input on the left [PM01].

as it is done by every other 3D graphics pipeline (see Figure 2.9). As already mentioned the inputs for this system are general 2D image maps, which can be either drawn or done by scanning statistical and geometrical maps, and in addition, the user can change different parameters which affect the produced virtual roadmap. A usual L-Systems is a parallel string rewriting mechanism based on a set of production rules [PM01, p. 303]. Each string is composed of multiple modules which then define the commands. Commands are then the actions that are taken further on to generate different structures. Parameters and constraints which belong to these commands are in turn stored in the modules. This leads to the problem that this approach can not be easily extended, as with growing constraints the production rules grow immensely. That is the reason why Parish et al. use extended L-Systems, which only generates a generic template at each step using L-Systems, meaning, by outsourcing the setting and modification of parameters into external functions. The generic template, also called *ideal successor*, then only consists of the string's modules but with unassigned parameters instead. The outsourced parameters and constraints then get retrieved by calling the external functions, split into the *globalGoals* determining parameter values and into the *localConstraints* adjusting the parameters to fit these constraints. This allows the system to be easily extended, without having to worry about the increasing number of rules.

Next to the problem of extending production rules in large structures like cities, there is also the problem of slightly changing existing production rules in retrospect and because of this receiving unexpected changes in outcome. This is why Talton et al. propose an approach to solving this issue, by enhancing the most common procedural interpretations, which are formal grammars such as L-Systems and shape grammars [TLL$^+$11].
This is accomplished by a high-level specification, which has to be met by the production computed from the given grammar. Specifications can be based on different forms, which restrict the final procedural model, optimized over several productions approximating

[a]


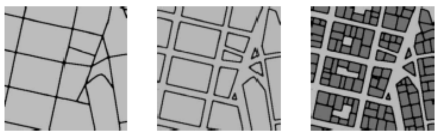
[b]


Figure 2.9: a) Pipeline from image maps to the rendered output. b) Left: Street map. Middle: Apartment blocks build by creating street crossings. Right: Generated Lots [PM01].

an ideal objective formed by the specification. These specifications can be formulated for example by a sketch, a volumetric shape, as well as an analytical objective. For constructing such an optimization Talton et al. use a generalized Markov chain Monte Carlo (MCMC) method, called Reversible jump Markov chain Monte Carlo [TLL$^+$11, p. 4], inspired by the original MCMC [GRS96].

These approaches solely concentrate on generating geometrical structures, but when it comes to city simulations other aspects get interesting as well, including the simulation of specific situations within the structure. Some examples are emergency situations, like the impact of floods on doors and windows, as well as visibility from certain locations on the map. This was also the thought behind CityGML by Gröger and Plümer in [GP12, p. 1], by interpreting semantic features and predicting an outcome to a situation.
To achieve this, CityGML assigns a semantic definition, attributes, relationships, and a 3D spatial representation to each feature represented in the model. Each feature relates to a module, which is categorized through their common attributes and the so-called core module, one example is shown in Figure 2.10 where the relation of a garage and its corresponding house is described by such modules. Furthermore, a terrain representation gets defined through the relief module, which is represented in different LoDs (Levels of Detail). This results in different modules like the building module, bridge module, tunnel module, and transportation module [GP12, p. 2]. The transportation module of CityGML allows constructing a roadmap, including railways, kerbstones, middle lanes, etc. to support route planning, especially for detailed truck route planning. One very interesting feature of CityGML is, that the different LoDs do not only affect the 3D representation of the models but also the semantic features, as most often these get less important or rich as they are further away from the user's viewpoint [GP12, p. 4]. As most specific applications require widely different features to simulate the desired outcome, CityGML provides the Application Domain Extension (ADE), making it possible to extend the

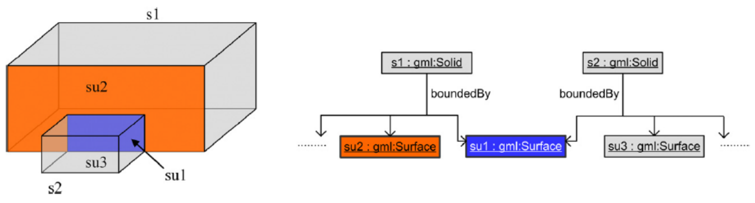


Figure 2.10: Topology Representation of a garage which shares common surface with the house and a resulting instance diagram showing the relationships between these objects [GP12].

given CityGML by defining new feature types, attributes, geometries, and associations [GP12, p. 10].

2.3.2 Interactive Urban Models

In chapter subsection 2.1.2, sketching was used to design 2D maps, which further on were used to generate structures procedurally. Another way to use sketching to your advantage is an approach proposed by Nishida et al. in [NGDA$^+$16], where rather than creating maps, entire buildings are sketched by the user. In this approach, simple procedural grammars are used as a foundation to turn sketches into realistic 3D models, and making use of a machine learning approach to solve the inverse problem, the best match to the sketch is found. For machine learning Nishida et al. train a convolutional neuronal network (CNN) with artificially generated data and use the resulting model to recognize and estimate the procedural rule intended by the sketch [NGDA$^+$16, p. 1] (see Figure 2.12). The goal is to combine the freedom of sketching and the detail-oriented procedural modeling, receiving the best of both worlds. It is not required to set up procedural rules manually and thus making it easier for the user to intuitively design a building, without having to think about the process behind the application. This means the user starts with a coarse model sketch and adds more and more detail to it, by sketching various object types e.g. windows, roofs, etc., which then get automatically defined by the already mentioned simple procedural grammar, so-called snippets (see Figure 2.11). After the procedural model is automatically generated by the CNN, consisting of various grammar snippets, the user is instantly able to visualize the 3D model in different rendering styles [NGDA$^+$16, p. 2]. Similar to the other applications the snippets get represented by split grammars, defined by a set of rewriting rules. These snippets are recognized/reviewed by the CNN every time the user draws new strokes, in the environment by using a mouse or a digital pen on a tablet, and added to the grammar, combining all snippets for the final 3D model visualization. This approach is then resulting in two CNNs, one for recognizing the snippets and one for estimating the parameters. The training of the CNNs takes place before the actual sketching and is then used as described for defining the model [NGDA$^+$16, p. 2-3]. Also, important to note is that not only one suitable model is given to the user, but different variations inspired by the original sketch [NGDA$^+$16, p. 2].

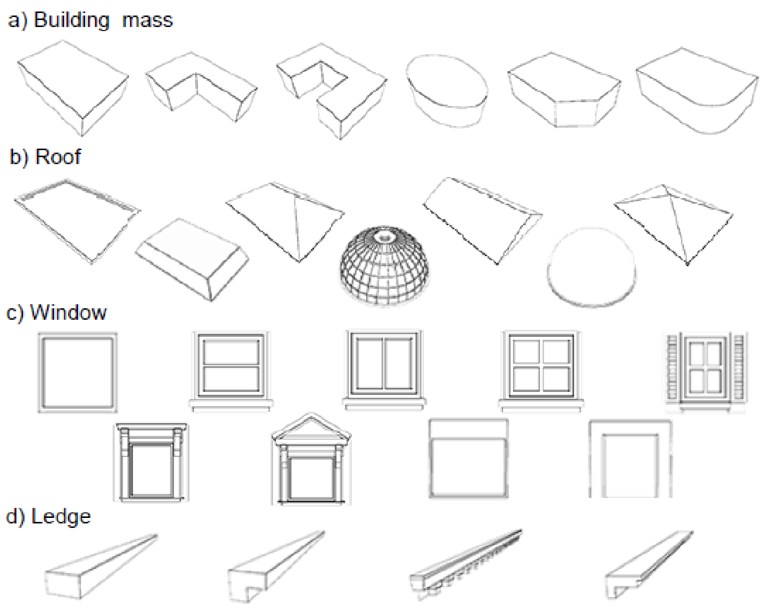


Figure 2.11: Snippet examples and Object types defined by parameters [NGDA+16].

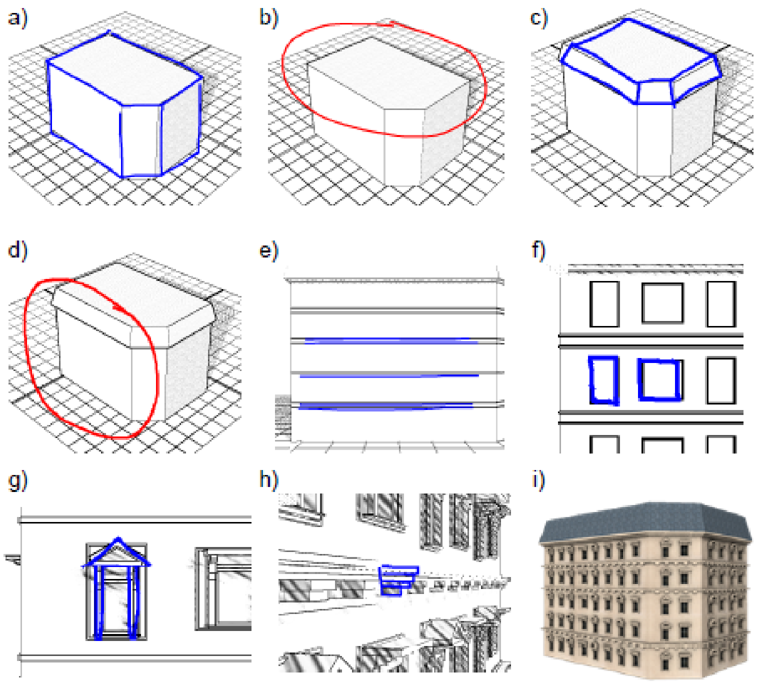


Figure 2.12: Sketching of a building, with each step creating a more detailed version of itself [NGDA+16].

2.4 Parsing Generators

The following paragraphs give a general definition of parser generators and procedural models. The two specific parser generators ANTLR and Bison will be discussed in more detail in the main part of this thesis.

Achieving a procedural model in the case of shape grammar is beginning with the construction, analysis, and recognition of grammatical input. This can be done via various methods, including hand-writing a parser and lexer yourself or using available parser generators making the development a little bit simpler. In this thesis, we are concentrating on the second method where some steps are already laid out to start analyzing your grammar. Parser generators in general use parsers and lexers to build the foundation on which code then gets generated and provide different features and guidelines to simplify the process. A parser specifies how the syntax of lexical input has to be analyzed, whereas a lexer specifies how to analyze the lexical components of a grammar input. Lexical analysis means recognizing patterns of multiple characters as so-called tokens, which are allowed in the structure of the grammar, and syntax analyzing stands for recognizing patterns in the composition of these tokens, also specified by the grammar structure. To recognize these patterns the lexer specifies tokens as the compositions of characters usually via regular expressions and the parser sets up rules including terminal and non-terminal symbols and also may specify actions that have to be taken if a rule applies. All of this then describes the grammar structure. These specifications then get converted into the target coding language, for example, Java, C, or C++, via the chosen parser generator. This constructed output then can analyze the grammar input, as it first gets analyzed by the lexer and parser individually and generates the desired actions to be taken further on, This could mean, for example, building a tree-like structure for further processing, as it is in our case.

CHAPTER 3

Overview

The following chapter gives an overview of Bison and ANTLR which are so-called parser generators and describes their advantages, disadvantages as well as their differences.

3.1 ANTLR

3.1.1 Definition

Developed in 1989 by Terence Parr, ANTLR is a parser generator, that constructs languages by building and walking parse trees modeled after a given grammar. These parse trees represent data structures and are automatically built through the parse generator, which is constructed by doing two important steps, lexical analyzing (Lexer) and syntax analyzing (Parser). This parse tree then can recognize and match the grammar to the input [Par14a]. The most recent version of ANTLR is ANTLR4 and generates a recursive-descent parser that uses an ALL(*) production prediction function [PHF14, p 3].

In recursive-descent parsers, we understand a top-down parser that builds a parse tree with the given grammar rules from the top down and left to right as it recursively parses the input. As ANTLR4 uses an ALL(*) production prediction function it eliminates the necessity of back-tracking. Back-tracking would be the procedure if an input symbol would not match the current grammar-production rule it back-tracks to the next grammar-production rule in order to match the input. But as ANTLR4 uses production prediction this means it predicts which grammar-production rule has to be used to replace the input via a look-ahead pointer pointing to the next input symbols. But to assure a back-tracking free predictive parser it only accepts LL(k) grammar, which is a type of context-free grammar, which means every sequence of production rules has to inherently lead to a terminal symbol in the end. LL(k) means also L for the left to right, L for the Left most derivation, and k is the number of lookahead symbols. This means ANTLR4 combines the well-known top-down LL(k) parsers with the GLR-like mechanisms and this is the so-called adaptive LL(*), or short ALL(*). Where GLR stands for Generalized LR, L again stands for the left to right and R for the rightmost derivation. GLR parsers are bottom-up parsers that process all possible production rules for an input to resolve the issue of grammars having more than one left-most derivation.

By moving the grammar analysis to parse-time, ALL(*) provides a solution to handle any non-left-recursive context-free grammar and supports direct left-recursion via grammar rewriting, which was not supported by the previous LL(*) parser as LL grammars usually do not handle left-recursion at all. Furthermore, ALL(*) is mentioned as GLR-like as it avoids the undecidability of a static LL(*) grammar analysis, by launching subparsers at specific decision points, one per alternative of a rule. This allows the parser to explore all possible paths, which then get dismissed if a path further down does not match the given input. Even ambiguous input can be resolved as the subparsers may coalesce together or reach the end of the file, as it resolves the problem in favor of the lowest production number [PHF14, p 1-2]. With this ANTLR4 accepts any context-free grammar as input, as long as it does not contain indirect or hidden left-recursion. The syntax is written with EBNF operators, a way to describe the syntax of context-free grammar, and token literals in single quotes [PHF14, p 3]. This in general is included in most parser generators, but what makes ANTLR more accessible are the many already included features, which do not have to be manually coded by the user.

3.1.2 Advantages

The following passage gives a more detailed description of the more general features, also used in this previous project, where ANTLR4 was used to generate parsing code from the grammar syntax input before switching to Bison, to in the end have a 3D model of a city be generated.

A really helpful feature is that ANTLR automatically generates a parser generator in the user-specified coding language with a given Lexer and Parser. This Parser from the beginning includes context methods, each belonging to one grammar rule and all the subrules included in this given rule. This makes it much easier to later on access all rules found in the grammar and the related subrules and values. The ready-to-go ParseTreeWalker can trigger events in Parse Tree Listeners, which then can invoke further actions specified by the user. The interfaces for these Parse Tree listeners are also automatically generated for every grammar rule and their implementations are either per default set to the base listener with empty implementations or by subclassing the base listener and overwriting implementations by user-specified code. Similar to Parse Tree Listeners, it is also possible to generate visitors, which differ from the Listeners as they walk the parse tree as specified by the user by themself and not only reacting to a ParseTreeWalker. This can be quite beneficial because it declutters the grammar as it detangles application-specific code and the grammar code [Par15]. Looking at the Listeners, ANTLR also comes with a BaseErrorListener which already includes multiple, different types of errors which can be, after setting up an ErrorHandler and Overwriting the necessary methods of the BaseErrorListener, then automatically recognized and invoke actions.

One very helpful feature, lying inside the parser generator files, which includes the rules and tokens of a specific grammar is the so-called *Options*, which change the way the code is generated and can make writing the parser generator simpler and more manageable. For instance, the grammar option *tokenVocab*, simplifies the Parser, as it automatically generates a tokens file and assigns token type numbers with the help of the Lexer and makes these tokens easily accessible to the Parser [Par16a]. This is specifically useful as the parser needs these tokens to build and describe the grammar production rules and otherwise would have to be generated and assigned manually. These mentioned automations the user to concentrate on the important parts, like writing the grammar itself, then on implementing an overwhelming and sometimes cluttered code that processes the grammar.

Although the ANTLR tool is based on the Java language, it is possible to generate parser generators in a good variety of language targets as the following, C++, C#, Python, JavaScript, Go, Swift, PHP, Dart, and of course Java. These can all be generated via a command line or Plug-ins of supported Development Tools [Par20]. With the ANTLR4 it is no longer necessary to fit the grammar to the underlying parsing strategy because it does grammar analysis dynamically at runtime with the ALL(*) production prediction function, rather than statically. This makes the coding way simpler and warnings, like reduce/reduce conflicts in Bison, won't give you a hard time anymore [Par14b, p. xi]. Furthermore, does ANTLR4 automatically turns ambiguous left-recursive rules into non-left-recursive rules as it sets the precedence equal to the order the alternatives of a rule appear [Par14b, p. 69].

One handy operation available in ANTLR lexer code is the option to skip specific matching tokens with *->skip* without having to think about line and/or character length skipping yourself, as the skip operation does it automatically. There are various operations like this in ANTLR, which simplify coding lexers and parsers, including the option to create different channels for specific token sequences, making it for example possible to skip comments in the main channel, but preserve the token stream in a hidden channel without losing the information, as it would happen with the skip operation [Par14b, p. 50].

Similar to static lexers, the ANTLR4 lexer builds a DFA (deterministic finite automaton), which according to different states in the automaton accepts or refuses certain input strings, but other than the static lexers is that ALL(*) lexers are predicated context-free grammars, which makes it possible to recognize context-free tokens like nested comments and gate them according to the semantic context [PHF14, p 4]. As stated in "Adaptive LL(*) Parsing: The Power of Dynamic Analysis" by Parr et.al. in 2014, ANTLR4 outperformed other parsers, including LALR(1), LL(k), LL(*) and PEG and was only 20% slower than the hand-build parser in the Java compiler itself [PHF14, p. 12].

3.1.3 Disadvantages

On the downside of ANTLR being an LL parser (Left to right with Left derivation), means it cannot handle left-recursive rules without the already mentioned automated change from left-recursive to non-left-recursive, which is the case in previous versions of ANTLR, like ANTLR v3 and still not possible with indirect or hidden left-recursions in ANTLR4. One performance issue in ALL(*) is the lookahead DFA cache, which is in line with the cost of GLL, which are generalized, left to right, leftmost derivation parsers, and GLR parsers, that do not reduce parser speculation. This means clearing the DFA cache before parsing is important as it can slow down the parse time [PHF14, p. 12]. Another side-effect of using any parser generator, in general, is that the user generally is limited by the tool and has to adjust accordingly, which would not happen with handwritten parsers as you have control over all your choices, but on the contrary takes more time and may be more difficult to create.

3.2 Bison

3.2.1 Definition

Bison, developed under the GNU Project and originally written by Robert Corbett is an open-source general-purpose bottom-up parser generator that creates a deterministic LR or generalized LR (GLR) parser with the Input of annotated context-free grammar. It uses LALR(1) parser tables, different from ANTLR4 which uses ALL(*). It is based on the Linux operating system and the programming language C. It is possible to generate parser generators in other languages, but it is most definitely more complicated than in ANTLR4. These include C++ and Java as experimental features [Pro14b]. An earlier developed variant called yacc is often associated with Bison, because Bison was made upward compatible with Yacc, by Richard Stallman, meaning no change should be necessary to switch from Bison to Yacc as some people still may be more familiar with the Yacc parser [Pro21]. To use Bison as a complete grammar interpreter with parser and lexer, the fast lexical analyzer generator called Flex is often used in combination with Bison as a parser generator. Flex was developed by Vern Paxson and similar to Bison, it is still known for its predecessor Lex. Flex is used for lexing in C and C++ and generates so-called scanners to identify tokens, also called lexical patterns in the input text [Est20]. Similar to ANTLR it has a parser and lexer which together generate a parser in the desired and available coding language. Different from ANTLR, Bison does not provide features in its base language tool, but more so concentrates on the lexical and analytical lexer and parser site. Bison does not provide an automatically generated parse tree, a walker, a Listener, or a Visitor, as it is the user's responsibility to code the desired data structure suitable to the input structure themselves. This gives the user a lot of freedom, but on the other side can take more time coding. In this project's case, this was interesting as different from the ANTLR implementation in the previous project version, where an automatically build parser tree was walked through via visitors to subsequently build an Abstract Syntax Tree (AST), it was possible to directly generate an AST via correct invocation of methods in the rules respective actions, without having to go through the tree a separate time again. This could be more difficult for someone without any experience in Bison, but as there are many resources and examples on the web, it is possible to learn fast.

3.2.2 Advantages

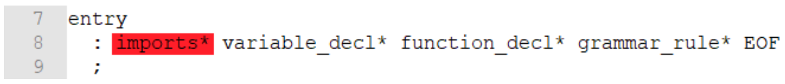
As mentioned Bison does not support too many features on the already generated site, but it does provide more freedom coding-wise. It is still widely used, as the resources and examples on the internet show. Bison is open-source and makes it possible to work freely on projects, as well as it is possible to contribute the own achievements to the Bison Gnu Project. Inverse to ANTLR, Bison uses LALR(1) grammars, which handles any left-recursion very well and is also preferred over right recursion, because this would use up space on the Bison stack, as all elements must be shifted onto the stack before the rule can be applied even once [Pro21, sec. 3.3.3 Recursive Rules]. This happens because, when the last token of a non-terminal is reached, it usually gets reduced, as Bison is a bottom-up parser [Pro21, sec. 5 The Bison Parser Algorithm]. But Bison does not always immediately reduce the rule when the last n tokens are reached and a rule matched because this would not be efficient as most languages would not be handled correctly this way. That is why Bison also does have lookahead tokens. This means if a reduction would be possible the parser may look ahead to make a decision [Pro21, sec. 5.1 Lookahead Tokens]. As Bison and Flex do not include too many features after the generation of the parser, they do include way fewer files and libraries to include than ANTLR does, this means it does take up less space. To start working with Bison and flex, users only need their executables, especially if they do not need custom build rules, as it is for Visual Studio, and for example, they only work with a command window. This sounds difficult but was making the process of testing easier in our case.

3.2.3 Disadvantages

Bison does not support features outside the parser and lexer and if for example specific data structures are desired, this has to be coded manually. As already mentioned Bison does not handle right-recursion very well as it uses up space on the stack [Pro21, sec. 5 The Bison Parser Algorithm]. To include a lexical analyzer it is necessary to either use for example Flex or even write your own scanner by hand, as Bison is not equipped with either [Pro21, sec. 1.8 Stages in Using Bison]. Shift/reduce and reduce/reduce conflicts can be pretty difficult to solve as it is not as simple to debug and find these problems. Error Handling and Error Recovery have to be implemented by the user and there is no automated way to solve these issues. It is important to understand the location handling in Bison as it gets important in error handling, skipping comments for example, and in general keeping track of the locations of the tokens. Generating a parser in a different language as the base language, may most of the time be more difficult than with ANTLR. Furthermore, as Bison and Flex are based on the Linux operating system it can be a little difficult to manage if a project's operating system is windows. This is because firstly testing the parser and lexer has to be done via a Linux compiler, for example, the GNU compiler called GCC. If the parser generation is not yet combined with the user's project, which from our experience working on the parser and lexer separately from the rest of the project, is at first easier to test and debug. Secondly, it can be quite difficult to find windows operating executables for Bison and in particular, Flex, if it is a priority to create a standalone application. Of course, it would be possible to build the binaries yourself, but this also was not as easy as we would have thought to achieve, at least with flex. On the other hand, if it is not important to have a standalone application this won't be a problem anymore, as the usual Bison and Flex binaries created, compiled, and dependent on the Linux compiler, would work just fine. But this means it would be dependent on for example Cygwin64 and its header files, although this would result in more occupied space for the project.

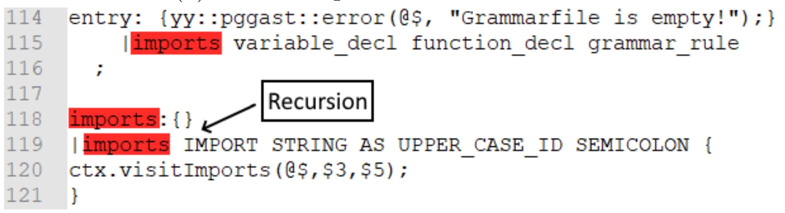
3.3 Differences

Besides the already mentioned differences in ANTLR and Bison, there are quite a few differences in the semantics of their parsers and lexers. At first glance, they seem quite similar, but with time these differences can build up fast when translating one into the other. First concentrating on the lexers, ANTLR and Bison switched the sides in describing the character compositions and token names. In ANTLR the token names stay on the left side, followed by a colon, and then finished with the character compositions and a semicolon. In contrast, Bison first describes the character compositions, followed by a space, and finished by returning the token names enclosed by brackets and a semicolon. As already mentioned an operation with the functionality like *->skip* in ANTLR is not present in Bison, but can be accomplished by either empty brackets or if location tracking is active, telling the location variable of Bison *yylloc* to take a step, which maintains the right location positions in the program but also ignores the character composition if it occurs. Bison can be quite tedious when it comes to describing equivalent compositions resulting in the same token name because in general such is described similarly as in ANTLR namely via /, but in Bison, there cannot be space anywhere between the different equivalents, as this results in an error. When it comes to returning or passing the value of tokens with the output to the parser, ANTLR simply does this automatically, whereas Bison has to be told what to return and which datatype it is. These datatypes unfortunately only support simple data types, like the common string, int, and so on. This comes important with the parser of Bison because it is necessary to declare, which data types can be passed on. Another operator depicted with a # can be really useful in ANTLR as it automatically, after compiling the parser EBNF grammar file, generates a rule context class definition for each label [Par16b]. This essentially means, it generates an interface from exactly on rule alternative, which gets called if the alternative matches the input. Differently, Bison demands to manually script the interface and call it via the method invocation, which further demands to create and include a context header file, which further down the line implements the interface. Some of the operators, most often recognized in REGEX, which is allowed in ANTLR do not get recognized in Bison, for



```
entry
  : imports* variable_decl* function_decl* grammar_rule* EOF
  ;
```

(a) ANTLR operation zero or more *



```
entry: {yy::pggast::error(@$, "Grammarfile is empty!");}
    |imports variable_decl function_decl grammar_rule
  ;

imports:{}
|imports IMPORT STRING AS UPPER_CASE_ID SEMICOLON {
ctx.visitImports(@$,$3,$5);
}
```

(b) Bison zero or more recursion

Figure 3.1: Compare the concept of "Zero or more" in ANTLR and Bison

example, the * operator, which usually means in REGEX zero or more of the Terminal or Non-Terminal stated in front. This makes it fairly difficult to just take ANTLR grammar and convert it into Bison because instead of using just a simple *, it is necessary to implement the function of this operator yourself. This can be done by for example invoking a recursion inside the rule/Non-Terminal which is allowed to be repeated zero or more times (see Figure 3.1), this also goes for the operator meaning zero or one time ? and also for the meaning of one or more with the operator +. With this Bison tends to have more coding lines, as this cannot be described with only one operator as in ANTLR and because of this sometimes simple Terminal symbols in ANTLR have to be converted to Non-Terminals in Bison to just make it possible to implement the meaning of one or more, zero or more and zero or one.

This is only the case in Bison, as in Flex it is possible to use these operators to describe a token because Flex does use REGEX to formulate the description of tokens.

CHAPTER 4

Method

The following chapter describes the steps we took to remove all ANTLR references and translate the parsing algorithm to Bison and Flex. To prevent misunderstandings the grammar files of the new Bison project version, which describe the rules and therefore the syntax of the given structure, are called Bison:Grammar_Parser, for the syntax grammar file, and Bison:Grammar_Lexer, for the lexical grammar file. The correspondent grammar files of the previous ANTLR version are called ANTLR:Grammar_Parser and ANTLR:Grammar_Lexer. When it is talked about grammar files in plural just the set of both files is meant. The corresponding context files used in Bison, which links the grammar input to a specific method creating the AST, are called Bison context header file and Bison context C++ file. Further on the C++ files generated by the grammar files, are called Bison:Parser_File and Bison:Lexer_File and ANTLR:Parser_File and ANTLR:Lexer_File when the corresponding ANTLR version is meant. Both of these files as a set are called Bison/ANTLR generated files. Files that have already existed before in the ANTLR version and are located within the C++ project will have the prefix PGG:<filename>. The input file which is fed to the generated files to create an AST is called input grammar file.

4.1 Project Structure

To get a better understanding of the project structure and organization this short chapter should give an overview of a very simplified version of the project and only parts which were important to our work, namely the parser generator replacement and GUI application. The overall project includes three important parts, the parser generator folder, the main project folder, and the GUI folder. The parser generator folder includes everything which is needed to generate an AST from the input grammar file given by the user. This is split into the grammar files, which either are written in Bison or ANTLR style grammar, constructing the rules and therefore the syntax of the grammar. These

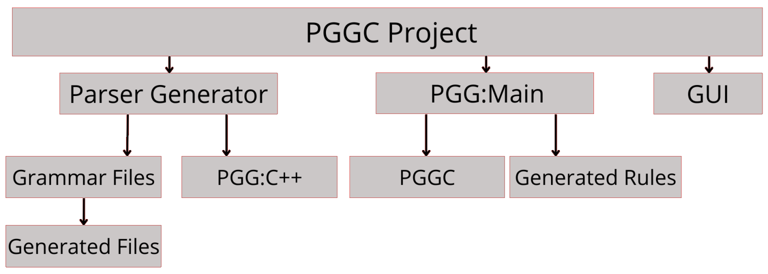


Figure 4.1: Project Structure Overview

grammar files, when compiled generate the generated files, which are the C++ equivalents to the grammar files. These then construct the AST. Inside the parser generator folder, there are also already previously existing C++ files, further on also called PGG:C++ which are used in the following method. Between the grammar files and the generated files there are also the context files, which mainly are used to connect the files to the methods of the PGG:C++ files. Next in the project are the main files, here called PGG:Main, which starts the process and calls the needed functions, for example, the PGG:C++ files and the generated files. These main files include two for us important parts, the PGGC part, and the generated rules part. The first uses the generated file from the parser generator to create an AST, the second uses this AST to build a 3D obj file, which then can be viewed in a visualization tool. Last we have the GUI folder which includes everything important to build the GUI, which is written in Phyton and accesses every before mentioned part of the project to preview the end result in a visualization technique chosen by the user. This should make it easier for the user to compile and execute the files, as it can be confusing. See this structure also in Figure 4.1.

4.2 Replicating previous project version and removing all references

For a short recap,the purpose of our project is to use a parser generator, in this case, ANTLR4 before switching to Bison, to generate C++ code from a given grammar syntax to create an AST which further on generates a 3D model of a city. As for our task, we had to remove all references of ANTRL to make it possible to integrate Bison instead, as well as Flex. To do this, it was important to first get to know ANTLR and with the help of "The Definitive ANTLR 4 Reference" by Terence Parr [Par14b], we explored what exactly ANTLR did and which files were generated in the process. To make a comparison possible and to check differences, access, and links between the previous ANTLR Version and the project we also kept an ANTLR Version in a separate folder to have easy access.

The first step was to remove the ANTLR library as it was easier to check the locations in the project files, where ANTLR was included. As it was expected, the most references to ANTLR are the files that are input and output files in the parser generation process, namely ANTLR:Grammar_Parser and ANTLR:Grammar_Lexer, as well as, ANTLR:Parser_File and ANTLR:Lexer_File and various other files already discussed in subsection 3.1.1.

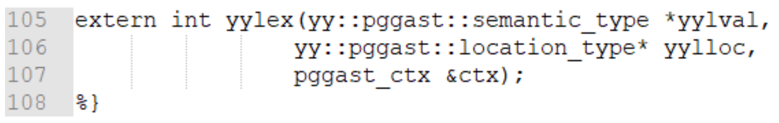
The ANTLR generated files can be just deleted ones, as they only get generated during the parser generation and are not fixed files and therefore do not have to be replaced as the new Bison:Parser_File and Bison:Lexer_File will be generated from the Bison grammar files as well. So the ANTLR grammar files were the first important files to rewrite into Bison grammar files because these do establish the foundation to generate the Bison generated files and their corresponding header files, which are used to build the interpreter for the given grammar rules and tokens, further on generating an AST.

4.3 Bison research and basic reconstruction

To get a good introduction of Bison, as well as in Flex we made great use of the book called "Flex & Bison" written by John Levine [Lev09]. This book explains Bison and Flex from the beginning and also gives a good example of an AST Tree structure, which was important in this project.

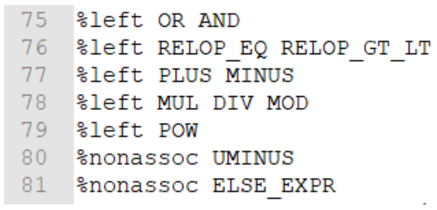
4.3.1 Reconstruction of the EBNF grammar files ANTLR:Grammar_Parser and ANTLR:Grammar_Lexer

The overall structure of ANTLR and Bison/Flex grammar rules and token description was fairly similar except, some already mentioned differences shown in section 3.3. Besides the semantic differences, Bison must tell the compiler, which language output to generate if the desired output is not C. As our project is in C++, the Bison grammar files were altered to fit this request. Because it is possible and simpler to leave the Flex-based Bison:Grammar_Lexer as a C-based file, it was left as it is, except for some little changes. Fortunately, Bison can use a C featured Bison:Grammar_Lexer output, and still can produce a C++ compiled parser generator. Therefore some additions have to be made in the Bison grammar files, as well as an additional context header file and if needed also a depending context C++ file, which describes the further application context and can be used in the next steps to connect the Bison generated files to the main project and even already built a functioning AST. First off the Bison grammar files have the altered file ending *yy*, which will then create C++ source and header files, as well as the header files *stack.hh*, *location.hh*, and *position.hh*, as long as not defined differently. Automatically a class called *yy::parser* gets defined by Bison, which includes the main parse routine, although this was altered via *%define parser_class_name {pggast}* to change it to a more fitting name for our project. Next, the coding language in which Bison:Parser_File should be written is declared by *%language "C++"* at the beginning of the Bison:Gammar_Parser, further on *%defines* creates the header and *%locations*



```
extern int yylex(yy::pggast::semantic_type *yylval,
                 yy::pggast::location_type* yylloc,
                 pggast_ctx &ctx);
%}
```

Figure 4.2: Declaration of the lexer function *yylex* to access in the parser file



```
%left OR AND
%left RELOP_EQ RELOP_GT_LT
%left PLUS MINUS
%left MUL DIV MOD
%left POW
%nonassoc UMINUS
%nonassoc ELSE_EXPR
```

Figure 4.3: associativity and precedence of operations in Bison

is specified for handling locations in the file. As Bison does not create a declaration of *yylex* that C++ requires, which is a function of the Bison:Grammar_Lexer to recognize and return tokens of the input to the Bison:Grammar_Parser, this is done by manually declaring yylex with Figure 4.2.

In the Bison:Grammar_Lexer file the already mentioned application context C++ file and the header file corresponding to the Bison:Parser_File are included to set the connection between the Bison generated files. Most interesting is the part where *YY_DECL* is defined because this declares the sequence yylex has to call, to match the Bison:Parser_File input [Lev09, p. 320]. To add additional arguments, which the Bison:Parser_File and yylex should accept, like the Bison:Parser_File class constructor, the declarations *%parser-param* and *%lexer-param* are defined with the arguments we want to add. As already mentioned, the Bison:Grammar_Parser does not automatically get the used tokens datatypes from the Bison:Grammar_Lexer and therefore have to be specified inside the *%union{...}* declaration. Only simple data types can be easily used to transfer between the grammar files like *int ival.* Inside of union not only the transfer data types are declared, but also every other datatype, which is used further on in the actions of the Bison:Grammar_Parser rules, this includes all datatypes specified in PGG:*ast.h*, among other things also struct definitions. This would then look like for example *std::map<int, std::string> *m;* or *struct MapEntry *me;*, where the datatypes names, here *m* and *me*, later are used to specify the return value of a rule/Non-Terminal via *%type <m> enclosed_curly_zero_or_one_input_string_mapping_entry;* . *%initial-action* sets the filename for the initial location in $, which is passed to the scanner and can be used to get the location when entering a rule [Lev09, p. 320]. As already discussed, ANTLR specifies directly inside of the rule any left-hand-side associativity Non-Terminals or right-hand-side associativity Non-Terminals with *lhs=* or *rhs=*, but in Bison, this is done beforehand and different to ANTLR, not the Non-Terminals get mapped, but the operation is specified left-hand-side or right-hand-side associative and the order of mentioning is also directly used to specify the precedence of operations.

4.4 AST construction and context file handling

Because the ANTLR Version of the AST creation is a little bit different and first constructs a parse tree, which gets converted into an AST Tree by a Visitor Pattern, it was necessary to find a way to reconstruct the end result with Bison.

In ANTLR the Visitor Pattern goes through the already constructed parse tree and saves the important information of the roots into struct objects, which on the other hand describe the AST structure, this is done via the files called PGG:*ast_builder.h/cpp*.

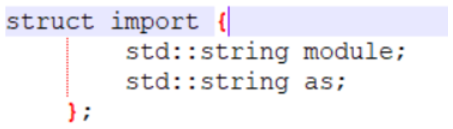
As already mentioned Bison does not create a parser tree automatically, but it can directly create an AST Tree if the right methods are called by the Bison:Parser_File which implements the AST building structure. At first, there was an attempt to use the files which already existed before translating the project from ANTLR4 to Bison, most importantly the already mentioned files PGG:*ast_builder.h/cpp*, which uses the automatically generated parse tree by ANTLR4 to build an AST, but just including the files in the Bison code was not working as expected, as the context C++ file would not recognize and find the given methods. Overall it was easier to directly generate an AST and include a routine in the Bison parser generator which does not build a parse tree first but uses a translated version of the PGG:*ast_builder.h/cpp* methods to directly create an AST out of the Bison parser generation.

The second try was to implement a similar version like in the book "flex & Bison" [Par14b] from John Levine under the chapter "An improved calculator that creates ASTs", which creates an extra file for all Bison included C++ methods and the main function starting the compiling process. This method could have worked but was unclear and messy. This may be because the version of the book works with C and not C++ files, because C++ Bison, as already mentioned previously, needs some alterations to function properly. This gave the idea to use the already existing context C++ file as was mentioned in subsection 4.3.1, which is used to translate a C-driven Bison compiler to a C++-driven one. This file already connects to the grammar files very well and also makes it easier to maintain the typical C++ class structure.

The methods needed to construct the AST Tree are already given by PGG:*ast_builder.h/cpp*, but had to be transferred into our now existing Bison context file, which easily can be called through the Bison:Grammar_Parser, furthermore, the methods of the PGG:*ast_builder* class had to be changed to work in this environment. But the outcome, by outcome meaning saving the AST information inside the struct patterns, declared in the AST file, still had to represent the same pattern as it is in the ANTLR version.

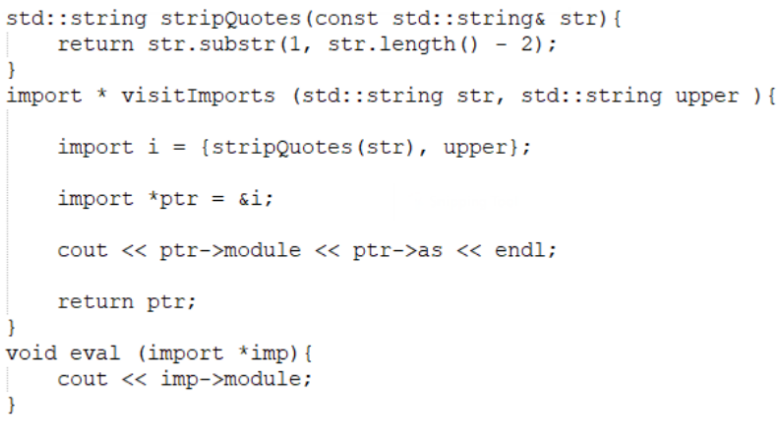
4.4.1 Construction of an AST Tree in Bison

In [Par14b] the construction of an AST Tree in Bison uses similar structures as well, like the file PGG:*ast.h* in our project. So it was natural to make use of this similarity and use the nearly same structs in the context file to create an AST Tree. These structs are important because they can represent some of the non-terminating tokens and also



```
struct import {
    std::string module;
    std::string as;
};
```

Figure 4.4: Simplified import structure



```
std::string stripQuotes(const std::string& str){
    return str.substr(1, str.length() - 2);
}
import * visitImports (std::string str, std::string upper ){

    import i = {stripQuotes(str), upper};

    import *ptr = &i;

    cout << ptr->module << ptr->as << endl;

    return ptr;
}
void eval (import *imp){
    cout << imp->module;
}
```

Figure 4.5: First evaluation and testing methods

rules to make sure they get correctly transferred between different levels of rules in the Bison:Grammar_Parser file. A simple struct with preferably easy methods was chosen to implement the first attempt and to create a simple output to test functionality.

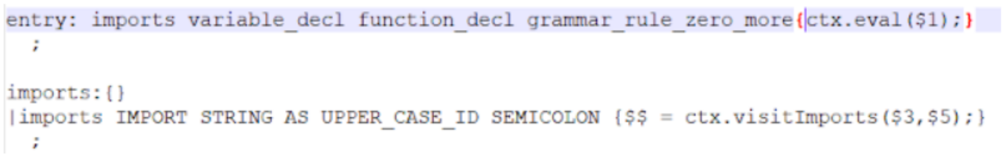
The structure used was a simplified version of the existing import struct as seen in Figure 4.4 and together with the first evaluation and test methods Figure 4.5, a very simplified connection between Bison and context files could be created.

These *stripQuotes* and *visitImports* methods were translated from the existing file PGG:*ast_builder.cpp* but were slightly rearranged to suit the testing. As one could see the output to the command line in *visitImports*, which only existed for testing.

The method *eval* is used to check the outcome of the import object when it is transferred from a deeper grammar rule to a higher one.

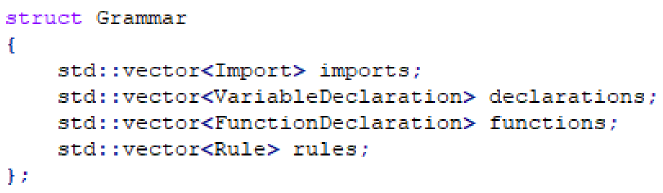
As already mentioned, all used structs were included in the Bison:Grammar_Parser file under *%union* and all rules and Non-Terminals have to get assigned a return value via *%type*. In Figure 4.6 you can see how these methods then get called inside the Bison:Grammar_Parser file as an action, enclosed by the curly brackets, of a specific rule.

Under the *imports* rule the assignment of the character *$$* is similar to the *return* call in a C++ method, which makes it possible to return a value, specified by the methods in the context file, here for example *ctx.visitImports*. This return value then gets returned to a higher level rule, here shown as the *entry* rule. Inside the *entry* rule, this value then can get accessed through the *imports* Non-Terminal, this is done by writing *$1*, which



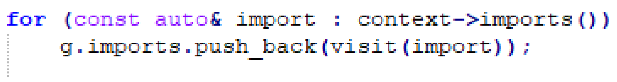
```
entry: imports variable_decl function_decl grammar_rule_zero_more{ctx.eval($1);}
  ;

imports:{}
|imports IMPORT STRING AS UPPER_CASE_ID SEMICOLON {$$ = ctx.visitImports($3,$5);}
  ;
```

Figure 4.6: Calling context methods inside Bison file



```
struct Grammar
{
    std::vector<Import> imports;
    std::vector<VariableDeclaration> declarations;
    std::vector<FunctionDeclaration> functions;
    std::vector<Rule> rules;
};
```

Figure 4.7: PGG:*ast.h* grammar struct



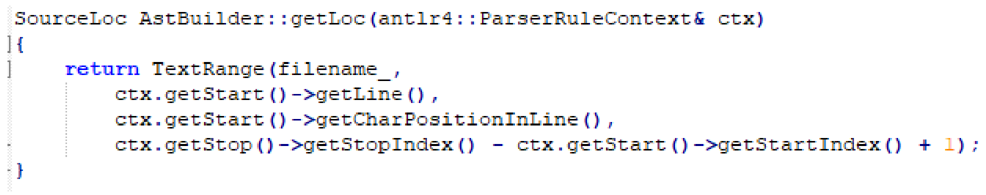
```
for (const auto& import : context->imports())
    g.imports.push_back(visit(import));
```

Figure 4.8: ANTLR visitor pattern example of *import* Terminal and *imports* Non-Terminal

accesses the first Non-Terminal or Terminal in a rule on the right side of the colon. If, for example, the fifth Non-Terminal wants to get accessed you would write *$5* instead. Then we just print out the transferred value of *imports* into the command line to check the result.

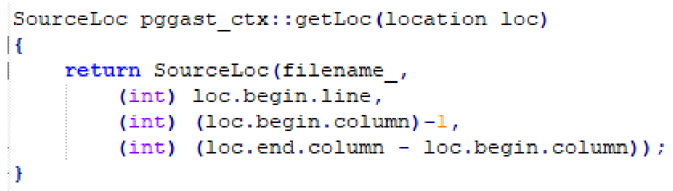
Now how is this turning into a struct, building an AST Tree? The Figure 4.8 shows one example of how the parse tree is walked through via the visitor pattern to save and construct the input syntax to an AST. Similar to this ANTLR routine the highest rule level values in the *entry* rule, *import*, *variable_decl*, *function_decl* and *grammar_rule* which after going through all rule levels are gathered and fit into their specific struct form, then get pushed back into a private variable inside the context files, formed like the *grammar* struct in the PGG:*ast.h* file Figure 4.7, where further down the line these values then can be accessed and used for further calculations and visualizations.

With this, it was possible to include all the other methods from the PGG:*ast_builder.cpp* and construct an AST Tree. The location tracking is important to mention, as some of the structures in PGG:*ast.h* are using a struct called *TextRange*, which gives information about the textual location of the used rule inside the Bison:Grammar_Parser file, especially filename, line number, character start position in line and character length of the rule. This is done a little differently than in ANTLR, as in ANTLR, the location can be directly accessed inside the PGG:*ast_builder.cpp/h* methods via another method (see Figure 4.9). But in Bison, it is necessary to first tell Bison that locations are needed as already mentioned in subsection 4.3.1, next the location can be accessed inside the rule action method enclosed by the curly brackets via *@$*. This returns the location of the whole rule grouping, but to access individual locations of Non-Terminals or Terminals



```
SourceLoc AstBuilder::getLoc(antlr4::ParserRuleContext& ctx)
{
    return TextRange(filename_,
        ctx.getStart()->getLine(),
        ctx.getStart()->getCharPositionInLine(),
        ctx.getStop()->getStopIndex() - ctx.getStart()->getStartIndex() + 1);
}
```

Figure 4.9: ANTLR location information



```
SourceLoc pggast_ctx::getLoc(location loc)
{
    return SourceLoc(filename_,
        (int) loc.begin.line,
        (int) (loc.begin.column)-1,
        (int) (loc.end.column - loc.begin.column));
}
```

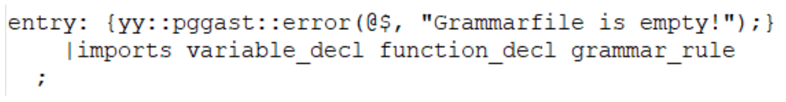
Figure 4.10: *getLoc* Method for extracting location informations

inside of a rule *@1*, *@2* and so on gives the exact locations. This then can be given as a parameter of a context file method, where further on the location information can be extracted in the method *getLoc(location loc)* Figure 4.10. This location information can be extremely useful for error handling, as the user would want to know for example where in the input grammar the mistake occurred to then alter the input to get recognized correctly by the interpreter.

4.4.2 Error handling and messages in Bison

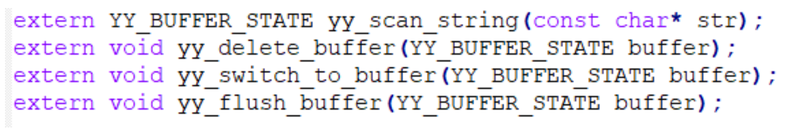
To have more control over the syntax error handling in Bison a custom error routine is created, which can be invoked via *%define parse.error custom* at the beginning of the Bison:Grammar_Parser file [Pro14b, Chapter 4.4.2]. But this means the *yy::report_syntax_error*, or in our case *pggast::report_syntax_error* has to be written additionally. The reason to write a custom syntax error method was to incorporate the file PGG:*error_handler* and its methods, which were already available and used in the ANTLR version of the project. So whenever a syntax error appears, meaning it does not fit into the grammar specifications, this method is called. In this method, the tokens, which do not fit into the grammar syntax are saved into an error message, as well as the expected tokens at this location, and further on this message then gets passed to the error handler including the location of the error. This entire *pggast::report_syntax_error* method is placed inside the Bison:Grammar_Parser file under namespace yy. To find the expected tokens another method of the parser context is called named *expected_tokens*. Not to mistake the parser context with the self-declared context file, as the parser context is "a type that captures the circumstances of the syntax error" [Pro14b, Chapter 10.1.6] and provides different already implemented methods like *expected_tokens*.

Every other error which may occur during the parsing process, but which is not handled within the syntax errors, can be manually defined inside the Bison:Grammar_Parser file



```
entry: {yy::pggast::error(@$, "Grammarfile is empty!");}
    |imports variable_decl function_decl grammar_rule
  ;
```

Figure 4.11: Non specified error example



```
extern YY_BUFFER_STATE yy_scan_string(const char* str);
extern void yy_delete_buffer(YY_BUFFER_STATE buffer);
extern void yy_switch_to_buffer(YY_BUFFER_STATE buffer);
extern void yy_flush_buffer(YY_BUFFER_STATE buffer);
```

Figure 4.12: Buffer definition

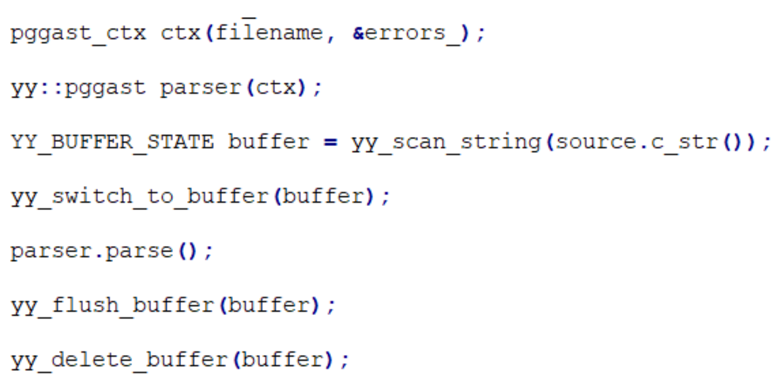
at a specific rule by calling another error method called *pggast::error*, which takes the location and an error message as parameters. This simply only passes the error to the PGG:*error_handler* file. In Figure 4.11 you can see an example of an error not covered by the syntax error handling, where if the whole grammar input file is empty and does not include any of the Non-Terminals here, the error method is called with a simple error message and the location of the error.

4.4.3 Connection to main project

To finally connect the Bison parser generator to our main project and execute via the main project, some simple lines had to be added in the file called PGG:*compiler.cpp*. First, the context file class had to be included at the beginning of the file and instanced inside the *parseInput* method, passing the grammar filename and the error handler. Next, the generated files here called *Parser.hpp/cpp* also had to be included and instanced, this time by passing the context we just instanced as well. Then different external buffer structs and methods for the Bison:Lexer_File input have to be defined (see Figure 4.12) Then a buffer has to be created after the parser instantiation passing the grammar input file as a string. Further, the buffer has to be switched to the just instanced one. To start the parsing simply *parser.parse()* is called Figure 4.13. After this, the buffer just has to be flushed and deleted again to free the space and the finished AST Tree then can be just accessed via the context file as already mentioned before.

At this point, the application worked fine, but was still not a standalone application, as Bison and Flex were still dependent on the Cygwin header files, as already mentioned in subsection 3.2.3. This could be passed over by using a prebuild executable of Bison and Flex for Windows called "Win flex-Bison", which is open to download at [Zho20]. To let the project then find these executables it is necessary to add them to the Makefile inside the PGG:*pgg_compiler* folder, which also now includes the new Bison files, and set the paths for the executables. This is necessary if the user should not have to change environment variables.

Because the main project is using the already generated files, the grammar files have to be compiled at the beginning also via the Makefile by writing a compile command for



```
pggast_ctx ctx(filename, &errors_);

yy::pggast parser(ctx);

YY_BUFFER_STATE buffer = yy_scan_string(source.c_str());

yy_switch_to_buffer(buffer);

parser.parse();

yy_flush_buffer(buffer);

yy_delete_buffer(buffer);
```

Figure 4.13: Connecting to the main project

these files and putting the now generated files into their designated folder inside the src folder, ready to use in the main project.

CHAPTER 5

Result

Eventually, the whole project should be easily accessible via a GUI executable, where it is possible to make changes to the input grammar file, create a new input grammar file and show the resulting 3D model by only clicking one button.

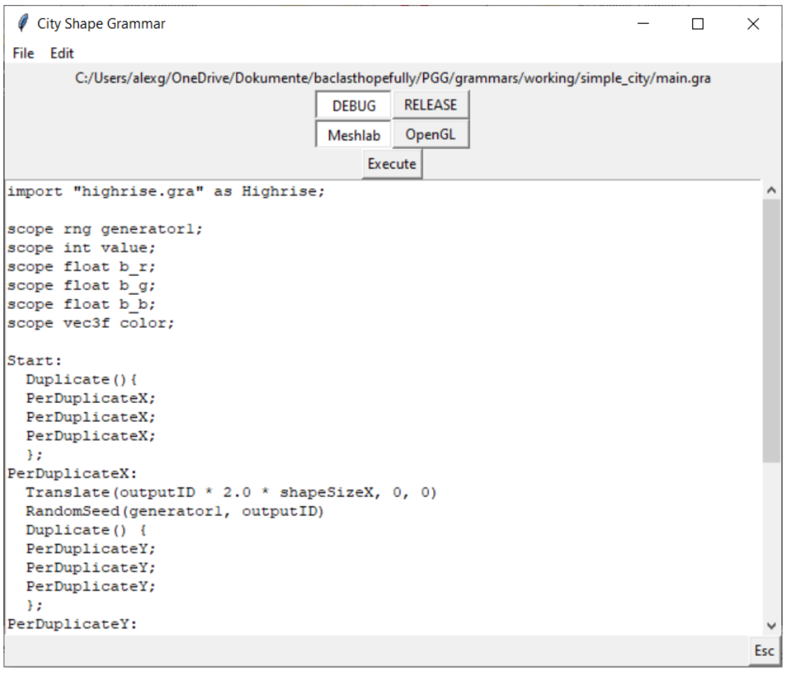
We made this possible, by writing a GUI application that is based on Phyton and a module called *tkinter* to model the GUI surface. For the 3D visualization, we first used an open-source OpenGL structure by pygame [obj], which we changed to fit our project, this turned out to be very slow for big city renders, so we also included an option to automatically use Meshlab [CM] to show the end-result, which turned out to be much faster. In the following chapters, we will demonstrate both versions and will try to discuss problems, which occurred during the process.

5.1 GUI Design and Functionality

Taking most of the space in the GUI Figure 5.1 is the textbox, which includes two main paths the user can take to execute a grammar file.

Scenario 1

The user starts from scratch and with opening the application gets presented by a still empty textbox. This box then can get filled by the user and has to be saved beforehand to start the execution process. If the user forgets to save the file, an explorer window is automatically opened for the user to save the file in a user-chosen directory. But if the user decides to close the saving window, a warning message box appears, because the file has to be saved to get executed. Was the file saved the execution process starts and further on the finished model is visualized, as long as the grammar file is correct.


City Shape Grammar
File Edit
C:/Users/alexg/OneDrive/Dokumente/baclasthopefully/PGG/grammars/working/simple_city/main.gra
DEBUG
RELEASE
Meshlab
OpenGL
Execute
Esc

```
import "highrise.gra" as Highrise;

scope rng generator1;
scope int value;
scope float b_r;
scope float b_g;
scope float b_b;
scope vec3f color;

Start:
  Duplicate(){
  PerDuplicateX;
  PerDuplicateX;
  PerDuplicateX;
  };
PerDuplicateX:
  Translate(outputID * 2.0 * shapeSizeX, 0, 0)
  RandomSeed(generator1, outputID)
  Duplicate() {
  PerDuplicateY;
  PerDuplicateY;
  PerDuplicateY;
  };
PerDuplicateY:
```

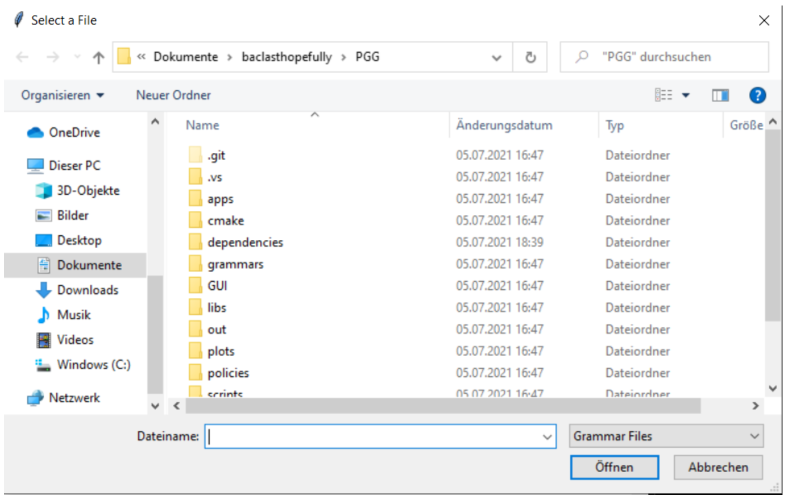
Figure 5.1: Complete GUI Application

Scenario 2

The user decides to use an existing grammar file, this is possible via the *File* tab (see Figure 5.3a) in the top left corner under *Import Grammar*, which opens up an explorer window Figure 5.2 to find and select an existing file from the computer. In this window, it is possible to either only search for grammar files with the ending .gra Figure 5.3b or the ending .txt. It is also possible to show all filetypes, but if a filetype does not suit the specifications, meaning the grammar rules written out as text, the execution would throw an error. Has the user selected a suitable file, this then gets shown inside the editable textbox, where the user still can make changes to the file if desired. These changes, as long as the grammar file already existed beforehand, get automatically saved before running the execution process. But it is also possible for the user to save the file via the *File* tab Figure 5.3a under either *Save* or *Save as*, whereas the first is overwriting the preexisting file which was chosen and the second can be used to save the freshly edited grammar into a new directory and or new name. Similar to the file selection explorer window it is also possible to choose from different filetypes Figure 5.3b.

Editing

In both scenarios, it is possible to edit inside the textbox and for more flexibility, it is possible, similar to other text editors, to undo and redo what was written, either via key combinations *Ctrl - Z* for undo and *Ctrl - Y* for a redo or via the GUI under the *Edit* tab in the top left cornerFigure 5.3c. It is also possible to use Copy and Paste key



Select a File

« Dokumente › baclasthopefully › PGG

"PGG" durchsuchen

Organisieren ▾ Neuer Ordner

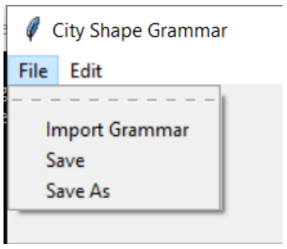
OneDrive, Dieser PC, 3D-Objekte, Bilder, Desktop, Dokumente, Downloads, Musik, Videos, Windows (C:), Netzwerk

Name	Änderungsdatum	Typ	Größe
.git	05.07.2021 16:47	Dateiordner	
.vs	05.07.2021 16:47	Dateiordner	
apps	05.07.2021 16:47	Dateiordner	
cmake	05.07.2021 16:47	Dateiordner	
dependencies	05.07.2021 18:39	Dateiordner	
grammars	05.07.2021 16:47	Dateiordner	
GUI	05.07.2021 16:47	Dateiordner	
libs	05.07.2021 16:47	Dateiordner	
out	05.07.2021 16:47	Dateiordner	
plots	05.07.2021 16:47	Dateiordner	
policies	05.07.2021 16:47	Dateiordner	
scripts	05.07.2021 16:47	Dateiordner	

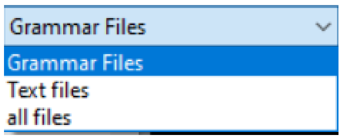
Dateiname: Grammar Files

Öffnen Abbrechen

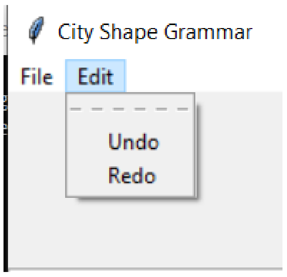
Figure 5.2: Import/Select File



(a) File tab



(b) Filetype selection



(c) Edit tab

Figure 5.3: Different GUI tabs and File type selection

combinations, as defined in most text editors as *Ctrl - C* and *Ctrl - V*.

5.1.1 Preexcution Options

As seen in Figure 5.1 there are also multiple options to choose from before executing the grammar. First, it is possible to run the project in debug and release mode, this is possible via a radio button on top of the GUI and as usual, takes a different amount of time to execute. Next, it is now possible to choose between the Meshlab visualization and the OpenGL half self-implemented visualization, which would take the user to two different windows of visualization.

5.1.2 Execution

Below that, the execution button is presented, which does all the compiling and executing which otherwise had to be done manually. It is no longer necessary to give the execution process execution commands as it is in the Visual Studio environment because the Phyton application gets this information itself via the file and the folder it originates from. During this execution process, first, the Visual Studio build is searched, and using this an MSBuild command is started with the debug or release configuration applied, to compile and further on executing the PGGC solution, which is the Visual Studio solution including the grammar interpreter and AST building process in which we worked on the most in this project. If this does work out without any errors, the same is done for the *generated_rules* solution, this builds the description which then results in an obj file, and in turn, describes the 3D model. It also builds the material mtl file, describing and creating materials from the present png textures if given. If this also works out fine, the execution process itself is done. Next is the automatic visualization based on the chosen visualization technique.

5.1.3 Error Messages

If something does not work out in either the first or the second compiling and executing process, an error message is shown appropriate to the place it occurred. For example "Error running Generated Rules" gives the place where the error occurred and additionally the thrown message of trying to run the solution is shown as well, which would mean something in the second execution (not compiling) process went wrong. All these errors get shown as a message box to alert the user instantly. For testing and monitoring reasons different messages also occur behind the application inside a shell, which shows where in the process the application is currently in. For example, if the grammar is finished being interpreted after running the PGGC solution it says "Finished grammar" in the shell. As said, this is mostly for debugging/testing reasons.

There are also some other error/warning messages shown, which occur in different scenarios, including syntax errors, empty file errors, the already mentioned not saved error, and an error if no *initGui* file does exist. The last one is important as this file should be generated automatically during the makefile process to save the source and binary directory for the GUI to know how to find its way around its necessary files. This should not happen but could occur if the file is deleted or something during the makefile process went wrong.

To then close the entire application the user can either click on the *X* in the top right corner or the *ESC* button on the GUI application, not to confuse with the escape key on the keyboard.

5.2 Visualization

Previously either Meshlab or OpenGL Visualization got chosen, but this doesn't mean it is not possible to switch again. To switch the user just has to select the other visualization and push execute again. When this is done the visualization window from before gets closed, so that no unnecessary amount of windows are pilling up.

5.2.1 OpenGL Open Source Visualization

Visualization

In the OpenGL Visualization (see Figure 5.4) the OpenGL API calls are written manually and everything which is needed to load and present an obj file is inside the files *objloader.py* and *objviewer.py* provided by pygame [obj]. These files and the setup of the visualization, including viewport, light conditions, obj directory, perspective, and more, are specified inside the file *main_GUI.py* shortly before the render loop. Inside of the render loop mouse and key settings are specified, as the model view can be rotated, translated, and zoomed inside the visualization window. Rotation is done via holding down the left mouse button and moving the mouse, translation without rotating the model is done by holding down the right mouse button and moving the mouse, and zooming is done by using the scrolling wheel on the mouse. The window then can be closed by either clicking the *X* in the top right corner or via the escape key *ESC*. It is a simplified preview of the model as the textures do not look as good as in Meshlab. Also, underlying colors do not look as it does in Meshlab.

Problems

The biggest problem here is, that if the models get too big, the visualization and mouse control get very laggy and slow, which often makes it impossible to get ahold of the whole model. Most possibly, this occurs because of not minimizing and perfecting the OpenGL API calls, which if called to often cause such speed issues. Another problem is that the visualization always starts inside of the model, as it is bigger than the anticipated view, this could be hard fixed with changing the viewport, but as every model differentiates in size, this would be a partial solution. If possible this should be made dependent on each model.

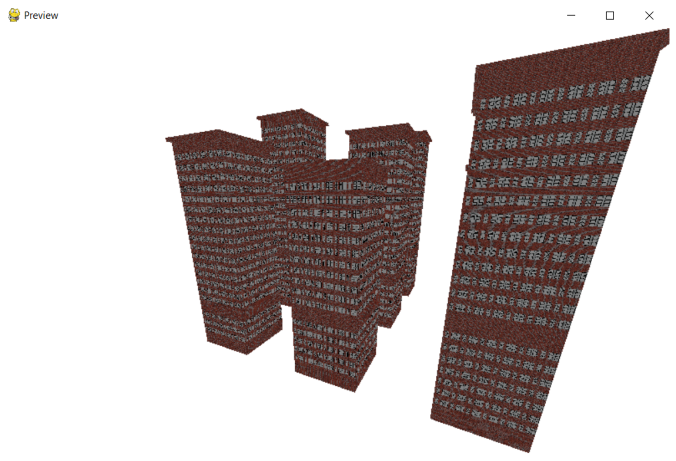
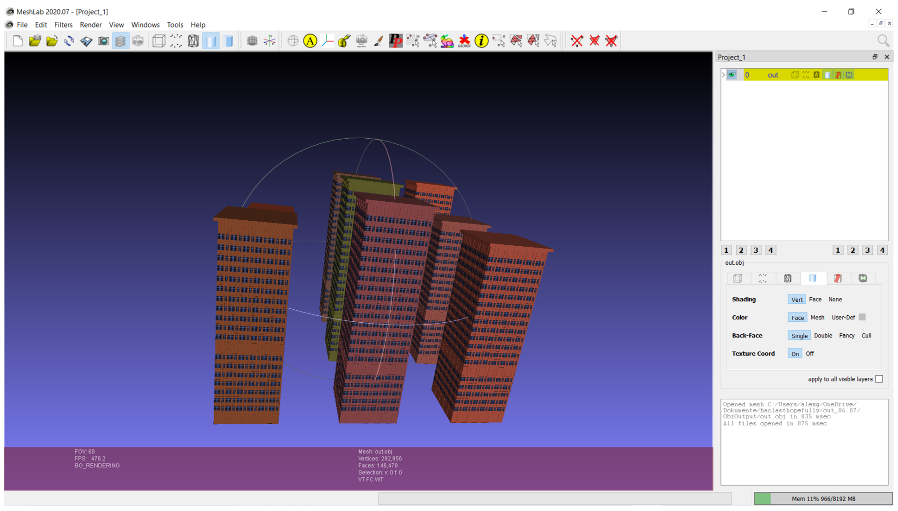


Figure 5.4: Example of a 3D city model displayed by the OpenGL Visualization

5.2.2 Meshlab Visualization

Visualization

Different from the OpenGL visualization, Meshlab Figure 5.5 is a more complex and completed open-source system to manipulate and visualize different model types [CM]. It is more perfected and has until the time of this writing no problems with the bigger models. It also is faster in showing the result and is implemented inside of the render loop via a second thread, which calls Meshlab with the according obj model via command line. It also displays the underlying colors, besides the textures, and starts the view on top of the model, zoomed out dependent on the model's size, which it should preferably do. For the controls, the rotation and zooming are done the same way as in the OpenGL visualization, and translation is done by holding the scrolling wheel down and moving the mouse. The controls are smoother than the OpenGL Visualization and on the whole do a better job, as it includes more options like wireframe mode, displaying the bounding box, various editing choices, and much more. The only downside to this was, that it is necessary to include the Meshlab executable and many different dlls which are needed for the application to work, this makes the project a little bit bigger but is not overwhelmingly bigger.


MeshLab 2020.07 - [Project_1]
File Edit Filters Render View Windows Tools Help
Project_1
0 out
1 2 3 4
1 2 3 4
out.obj

Shading [✓ Vert] Face None
Color [✓ Face] Mesh User-Def
Back-Face [✓ Single] Double Fancy Cull
Texture Coord [✓ On] Off
apply to all visible layers ☐

```
Opened mesh C:/Users/alexg/OneDrive/
Dokumente/baclasthopefully/out_06.07/
ObjOutput/out.obj in 835 msec
All files opened in 875 msec
```

```
FOV: 60
FPS:  476.2
BO_RENDERING
```

```
Mesh: out.obj
Vertices: 292,956
Faces: 146,478
Selection: v: 0 f: 0
VT FC WT
```

Mem 11% 966/8192 MB

Figure 5.5: Example of a 3D city model displayed by the Meshlab Visualization

CHAPTER 6

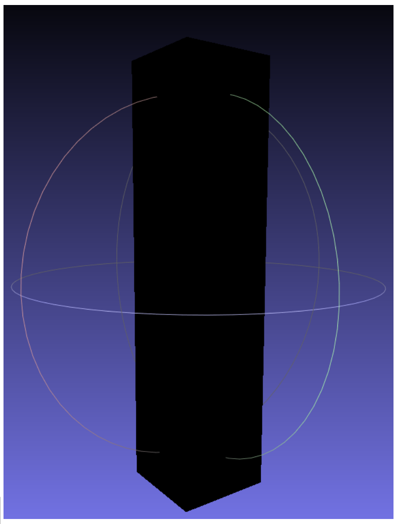
Evaluation

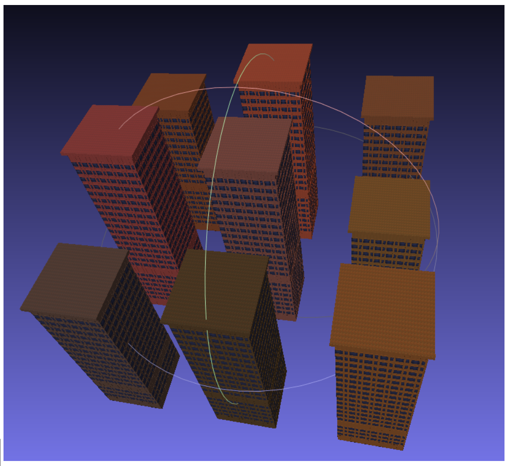
This chapter should give a summary of the results, including performance and memory consumption which were achieved by the prior versus the current project. As well as the differences in the visualization techniques used in the current project. In these following cases, an NVIDIA GeForce RTX 2070 GPU is used. For the comparison, itself, different in complexity varying grammar files are used in Debug mode. These included a very simple structure of a high-rise, a simple city structure, a bit more complex city structure, a complex forest structure, and a very complex Hilbert Cube, for a good representation in different structural styles. See Figure 6.1 a, b, c, d, and e for the resulting structures in Meshlab.

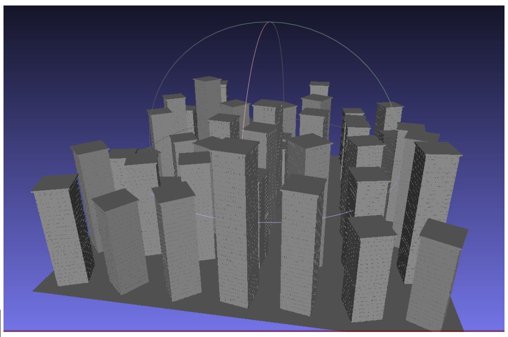
When looking into the performance in ANTLR and Bison there is not a huge difference in performance in the compilation of the PGGC part, which is the location we worked on the most and is responsible for generating a C++ AST with the grammar file as input. The average difference in seconds is around 0.36 seconds, with around 0,2677 seconds as the lowest difference compiling simple highrise and with around 0,43 seconds as the highest difference compiling the city structure. In this case, Bison is just a little bit faster than ANTLR. For a more detailed overview see Table 6.1. Of course, the difference in performance varies between every run, but the average difference is around the same every time.

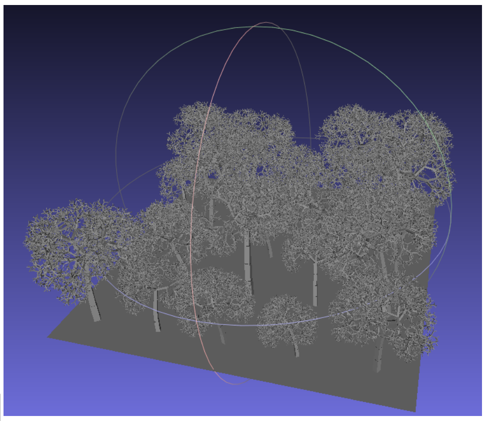
Surprisingly the discrepancy starts to show in the compilation of the generated rules, which is responsible for using the AST grammar output of the PGGC compilation to generate the 3D obj file. As in the ANTLR version, the compiling time most of the time is higher, especially when looking into the more complex structures. Bison is then under a second faster for simple structures but for more complex ones, it is around 30 seconds faster than ANTLR, which you can see in Table 6.1.

The memory consumption difference in Bison and ANTLR in *pgg_compiler* folder, which includes the file used to generate the C++ AST for PGGC compilation, is in Bison


[a]


[b]


[c]


[d]

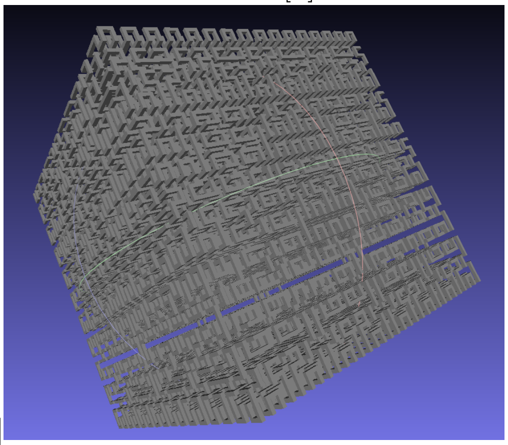

[e]

Figure 6.1: Different test structures ordered after complexity. a) a simple high-rise depicted as a simple box, b) a simple city with included materials/colors, c) a little more complex/bigger city d) a complex forest structures, with multiple small branches and e) a very complex Hilbert Cube.

Grammar	Performance Bison in sec			Performance ANTLR in sec		
	ALL	PGGC	GR	ALL	PGGC	GR
city	67.54	2.08	65.45	82.88	2.52	80.36
forest	72.82	2.08	70.74	87.94	2.46	85.47
highrise	17.68	2.63	15.04	17.58	2.59	14.99
Hilbert	84.48	2.07	82.40	106.67	2.50	104.17
quad_roof_test	18.76	2.11	16.65	28.66	2.51	26.15
residence	18.97	2.13	16.84	19.02	2.38	16.64
shape_grammar_basics	18.38	2.12	16.25	18.62	2.34	16.27
simple_city	26.66	2.19	24.46	29.78	2.46	27.31
simple_highrise	19.13	2.11	17.02	19.17	2.38	16.79
sponge	123.93	2.04	121.88	159.86	2.41	157.44
testing	90.05	2.11	87.94	116.02	2.67	113.34

Table 6.1: Table of ANTLR and Bison performance in seconds. First column of each performance shows the whole compilation time, third the compilation time of PGGC and last the compilation time of generated rules. The very first column shows the grammar name.

around 1,09 MB less than the ANTLR Version. When it comes to the whole project memory consumption, the new project is around 205,6 MB bigger, mostly because of the now available GUI application which takes up to 161 MB. This is partially the case because two visualization techniques are present and take up memory space which will also be elaborated further on.

The completed execution file of PGGC in ANTLR is 4,76 MB and of the generated rules it is 425 KB. For Bison, the PGGC execution file is 2.07 MB and for the generated rules it is 697KB.

For the performance test of Meshlab and OpenGL, the time between pushing the execute button and the finished visualization preview display is measured. This means including compilation time of before mentioned PGGC and generated rules.

The performance of the simpler objects like simple highrise and simple city (see Figures 6.1 a and b) do not differ as much yet, as it with the more complex structures. Where surprisingly die OpenGL version is faster for the simple highrise with around 22 seconds, whereas Meshlab needs around 29 seconds. But from the simple city on, Meshlab is faster with around 32 seconds and OpenGL with around 50 seconds.

When going further to a just little more complex structure as with the city (see Figure 6.1 c) you can already see a bigger difference in performance as the visualization with Meshlab takes around 96 seconds, whereas the OpenGL visualization takes around 210 seconds.This already gives a hint about the performance discrepancy, which is going to get bigger with the complexity of the structures.

With the example of the forest (see Figure 6.1 d), it is interesting to observe, that Meshlab is taking less time than for the city structure with about 91 seconds, but OpenGL takes more time with this kind of structure, resulting in a performance of way over 200 seconds.

Lastly, the Hilbert cube hits the highest time consumption with Meshlab with 107 seconds, which still is lower than the worst performance with the OpenGL version. The OpenGL performance on the Hilbert cube is around 257 seconds.

This concludes the observation of the different visualization types and underlines the fact that Meshlab is more efficient performance-wise when visualizing just a little more complex structures.

When looking into the memory consumption of both visualizations in the current state, Meshlab takes up around 122 MB with multiple libraries and dlls to make the standalone version of our project work. Whereas OpenGL lies by only 7,89 KB as it only uses two simpler Phyton files.

CHAPTER 7

Conclusion

After finishing this project there is still room for improvement, as the obj model generation still takes lots of time. As it does not only have to compile two files but also execute both, which can take up lots of time, especially the second round where the generated rules get transformed into the obj file inside the *generated_rules* solution. This could be made faster if it would not be necessary to execute two files but to combine and simplify it into just one execution. As well as for the OpenGL Visualization, this could be done smarter, as discussed, by minimizing API calls and doing most of the calculating on the GPU side, which would take less time to compute. For the future, it would be interesting to use this project for game and simulation development, where not only the city would get generated automatically, but also include collision detection to make it more accessible for simulations like crowd simulations, flooding simulations, sound travel simulations, and much more. In this thesis we described and compared parser generators, their advantages, and disadvantages, giving a view over techniques to build and create an automatic 3D modeling system and showing my simple GUI application which should make the process of modeling a city simpler for the user. But as the project and especially the GUI is now, it is still very plain and does not include many editing possibilities. As for the future, it would be very interesting to not only simplify the executing process as it is done now, but to also include a system, for example, different settings which inherit values important for the generating process, which would make the user being able to create a 3D model, without even having to know the grammar rules and structure, but only has to fill out given specifications, like how many buildings, building height range, grid, and much more. This could make the program more accessible to users which do not have specific knowledge in this field and would possibly shorten production time even more. Additionally to this idea, it would be interesting to implement a real-time solution, where the preview 3D model is changed as the settings are altered, as it could give a better understanding of how the generating process works depending on the settings.

List of Figures

Glossary

ecotope An area of homogeneous terrain and features. 6

Bibliography

[BWS10] Martin Bokeloh, Michael Wand, and Hans-Peter Seidel. A connection between partial symmetry and inverse procedural modeling. In *ACM SIGGRAPH 2010 Papers*, SIGGRAPH '10, New York, NY, USA, 2010. Association for Computing Machinery.

[CM] Paolo Cignoni and Alessandro Muntoni.

[DHL$^+$98] Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomír Měch, Matt Pharr, and Przemyslaw Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. *Proceedings of the 25th annual conference on Computer graphics and interactive techniques - SIGGRAPH 98*, 1998.

[Est20] Will Estes. The fast lexical analyzer - scanner generator for lexing in c and c++, sep 2020.

[GP12] Gerhard Gröger and Lutz Plümer. Citygml – interoperable semantic 3d city models. *ISPRS Journal of Photogrammetry and Remote Sensing*, 71:12 – 33, 2012.

[GRS96] W. R. Gilks, S. Richardson, and D. J. Spiegelhalter, editors. *Markov chain Monte Carlo in practice.* London: Chapman & Hall, 1996.

[Lev09] John Levine. *Flex & Bison: Text Processing Tools.* " O'Reilly Media, Inc.", 2009.

[Lin68] Aristid Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280–299, 1968.

[MM11] P. Merrell and D. Manocha. Model synthesis: A general procedural modeling algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 17(6):715–728, 2011.

[MP19] Ignacio Martin and Gustavo Patow. Ruleset-rewriting for procedural modeling of buildings. *Computers & Graphics*, 84:93 – 102, 2019.

[MWH+06] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, page 614–623, New York, NY, USA, 2006. Association for Computing Machinery.

[NGDA+16] Gen Nishida, Ignacio Garcia-Dorado, Daniel G. Aliaga, Bedrich Benes, and Adrien Bousseau. Interactive sketching of urban procedural models. *ACM Trans. Graph.*, 35(4), July 2016.

[obj] Objfileloader - wiki.

[Par14a] Terence Parr. About the antlr parser generator, 2014.

[Par14b] Terence Parr. *The Definitive ANTLR 4 Reference.* The Pragmatic Programmers, LLC, second edition, 2014.

[Par15] Terence Parr. Parse tree listeners, dec 2015.

[Par16a] Terence Parr. Options, march 2016.

[Par16b] Terence Parr. Parser rules, Nov 2016.

[Par20] Terence Parr. Runtime libraries and code generation targets, may 2020.

[PHF14] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive ll(*) parsing: The power of dynamic analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &amp; Applications*, OOPSLA '14, page 579–598, New York, NY, USA, 2014. Association for Computing Machinery.

[PL90] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. The algorithmic beauty of plants. *The Virtual Laboratory*, 1990.

[PM01] Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, page 301–308, New York, NY, USA, 2001. Association for Computing Machinery.

[Pro14a] GNU Project. Gnu bison, 2014.

[Pro14b] GNU Project. Gnu bison, 2014.

[Pro21] GNU Project. *Bison 3.7.6.* Free Software Foundation, Inc, 3.7.6 edition, mar 2021.

[STdB11] R.M. Smelik, T. Tutenel, K.J. de Kraker, and R. Bidarra. A declarative approach to procedural modeling of virtual worlds. *Computers & Graphics*, 35(2):352 – 363, 2011. Virtual Reality in Brazil Visual Computing in Biology and Medicine Semantic 3D media and content Cultural Heritage.

[TLL+11] Jerry O. Talton, Yu Lou, Steve Lesser, Jared Duke, Radomír Měch, and Vladlen Koltun. Metropolis procedural modeling. *ACM Trans. Graph.*, 30(2), April 2011.

[TS13] Bojan Tepavčević and Vesna Stojaković. Procedural modeling in architecture based on statistical and fuzzy inference. *Automation in Construction*, 35:329 – 337, 2013.

[VGDA+12] Carlos A. Vanegas, Ignacio Garcia-Dorado, Daniel G. Aliaga, Bedrich Benes, and Paul Waddell. Inverse design of urban procedural models. *ACM Trans. Graph.*, 31(6), November 2012.

[WWSR03] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. *ACM Trans. Graph.*, 22(3):669–677, July 2003.

[Zho20] Alex Zhondin. Win flex-bison, Jun 2020.