U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Improving Expressiveness, Integration and Manageability in Procedural Content Generation

Pedro Amorim Brandão da Silva

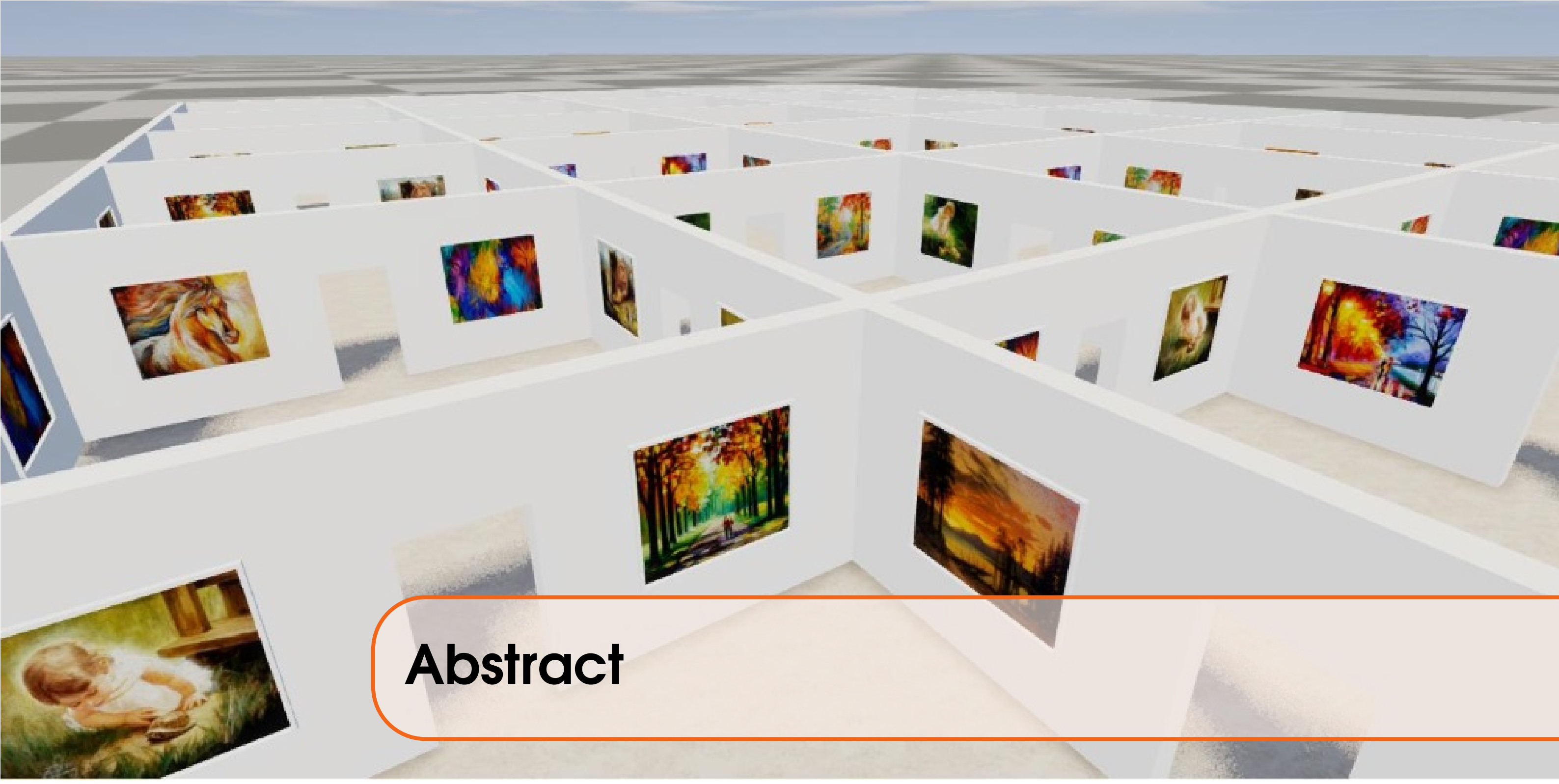
Doctoral Program in Informatics Engineering

Supervisor: António Coelho
Co-Supervisor: Rafael Bidarra

This work has been supported by the Portuguese government, through the National Foundation for Science and Technology - FCT (Fundação para a Ciência e a Tecnologia) through the Ph.D. Scholarship SFRH/BD/73607/2010 and partially through the project PTDC/EIA-EIA/108982/2008 entitled "3DWikiU – 3D Wiki for Urban Environments".

Latex template was obtained from:
http://www.latextemplates.com/template/the-legrand-orange-book.

Dedicated to my family and friends.



Abstract

The size and complexity of virtual urban environments make their production through manual approaches a very expensive, tiresome and time-consuming endeavor. On the other hand, considering that urban environments are bound by certain guidelines and rules, *procedural content generation* arises as a solution to automate the building process by encoding their pattern-repetitive nature into computer algorithms.

There are many techniques for generating many urban environment features (e.g. terrains, roads, vegetation, buildings) and many types of content (e.g. models, textures, items, music, sound), in which human intervention is reduced to the definition of control rules. However, there are still many challenges concerning expressiveness and manageability of these rules, which has so far restricted the integration and, ultimately, the design possibilities of several types of content.

This thesis focuses on the definition of a new rule specification paradigm based on *Procedural Content Graphs*, a visual programming language that, using a graph-based representation, employs nodes to describe procedures for creation, transformation, analysis or filter of content data and edges to describe the flow of that data between the nodes. This representation has proved to be more readable and manageable than other popular alternatives such as grammars and L-systems. This language has more expressive power than preceding approaches as it provides the means to generate and integrate diverse types of content in a single pipeline, as well as to produce structures and solve control issues that would otherwise be difficult or even impossible to achieve with existing alternatives.

In order to design and control procedural content graphs, a solution for procedural generation of virtual urban environments - *the Construct* - has been implemented. Built with extensibility in mind, this system provides a flexible application programming interface that allows the definition

of new procedure libraries, as well as plugins to introduce and enhance the visualization and manipulation possibilities of any type of content.

There have been very positive user reviews from conducted surveys and several public presentations on this work, acknowledging the improvement of the expressiveness and manageability in rule description. Both the described methodology and implementation have already been employed in several use cases and research projects, confirming the extent and suitability of this work within several domains, but especially in the development of virtual urban environments.

Resumo

A escala e complexidade dos ambientes urbanos virtuais são factores que tornam a sua produção, através de meios manuais, numa tarefa muito dispendiosa, cansativa e demorada. Por outro lado, considerando que os ambientes urbanos se encontram sujeitos a determinadas directrizes e regras, a *geração procedimental de conteúdos* surge como uma solução para automatizar o seu processo de modelação através da codificação dos seus padrões em algoritmos computacionais.

Existem muitas técnicas capazes de reproduzir muitos dos elementos que compõe um ambiente urbano (terrenos, estradas, vegetação, edifícios, etc.) e muitos tipos de conteúdos (modelos, texturas, objectos, música, som, etc.) na qual a intervenção humana necessária é reduzida à simples definição de regras de controlo. Contudo, existem ainda muitos desafios no que diz respeito à expressividade e facilidade de gestão destas regras, o que tem resultado em limitações na integração e nas possibilidades de concepção de diversos tipos de conteúdos.

Esta tese foca-se na definição de um novo paradigma de especificação de regras baseado em *Grafos Procedimentais de Conteúdo*, uma linguagem de programação visual que, usando uma representação de grafos, utiliza nós para descrever procedimentos de criação, transformação, análise ou filtro de conteúdos e arestas para descrever o fluxo dos mesmos entre os nós. Esta representação provou ser mais legível e fácil de gerir do que outras alternativas populares, tais como gramáticas ou sistemas L. Esta linguagem tem um maior poder expressivo que abordagens anteriores apresentadas na medida em que providencia formas de gerar e integrar diversos tipos de conteúdos numa única *pipeline*, bem como de produzir estruturas e resolver questões que de outra forma seriam difíceis ou mesmo impossíveis de alcançar com alternativas existentes.

Com vista a conceber e controlar os grafos procedimentais de conteúdo, foi implementada uma solução para a geração de ambientes urbanos virtuais - o *Construct*. Criado com vista à extensibilidade, este sistema providencia uma interface de programação flexível que permite a

definição de novas bibliotecas de procedimentos, bem como de módulos para a introdução e melhoria das possibilidades de visualização e manipulação de qualquer tipo de conteúdos.

Foram recebidos comentários e pareceres muito positivos de várias apresentações e questionários sobre este trabalho, reconhecendo a melhoria na expressividade e capacidade de manipulação das regras. Tanto a metodologia como a sua implementação já têm vindo a ser utilizados em diversos projectos académicos e de investigação, confirmando o alcance e aplicabilidade deste trabalho em vários domínios, em especial no desenvolvimento de ambientes urbanos virtuais.

Acknowledgements

I would like to start by thanking my supervisor, Prof. António Coelho, for the continuous support, diligence and sympathy throughout my Ph.D. study, for providing me several academic experiences, ranging from research projects to teaching opportunities, but most of all, for having introduced me to procedural content generation since my master thesis, a topic that has become my passion and the main motivation to pursue a Ph.D. enterprise.

I would also like to express my gratitude to my co-supervisor, Prof. Rafael Bidarra, for his expertise in the field, for his teachings on paper redaction and research practices, for his patience to deal with my stubbornness, as well as his enthusiasm in advertising my work to other people and environments. I also owe him a big thanks for his help with the preparation and aftermath of my 9-month stay to Delft, and for introducing me to a great number of interesting people and activities during my visit.

A very special thanks goes to all my colleges at the Esri R&D in Zurich, in particular to Pascal Müller, for the great internship experience, for their friendliness, time, attention and availability for extensive discussions. Having had this opportunity was definitely a major push in my work, and a determinant factor for the success of this thesis.

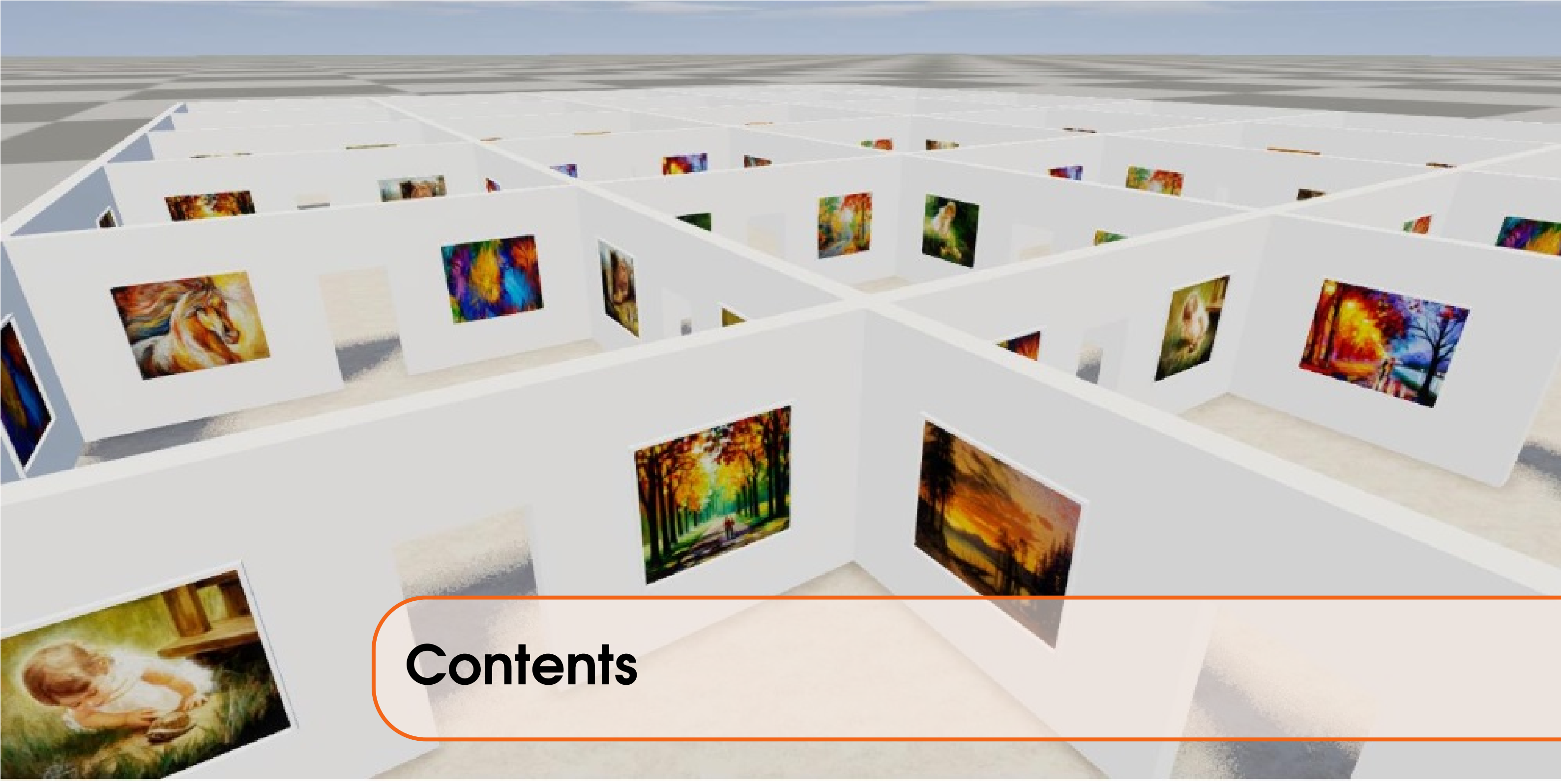
I would also like to thank the Faculty of Engineering of the University of Porto (FEUP), more concretely the Department of Informatics Engineering (DEI) for providing a workspace for the development of this work, to INESC Porto for acting as the accompanying institution and to the Foundation of Science and Technology (FCT) for the research grant SFRH/BD/73607/2010 that supported the development of this project. I also have to thank all the colleagues and professors at FEUP and the University of Trás-os-Montes and Alto Douro (UTAD) for the fruitful discussion sessions from which many ideas and lessons have resulted.

My gratitude goes also to the Delft University of Technology and the Computer Graphics and Visualization Group for accepting me as a guest visitor and for providing me with a friendly and passionate environment to work, learn and discuss new approaches and solutions. A special thanks goes to Prof. Elmar Eisemann for his persistence and enthusiasm in my work and its publication, and for finding time in his schedule for sharing new ideas and tips.

I have also to thank all Construct pioneers, i.e. all the students, professors and researchers that, even at a young and prototypical phase of my work, have believed in its merit and applicability, having developed their own projects, despite the lack of proper documentation. In this sense, a big thanks goes to Ruben Smelik and the TNO: Netherlands Organisation for Applied Scientific Research for their initiative and effort in hosting some of these projects.

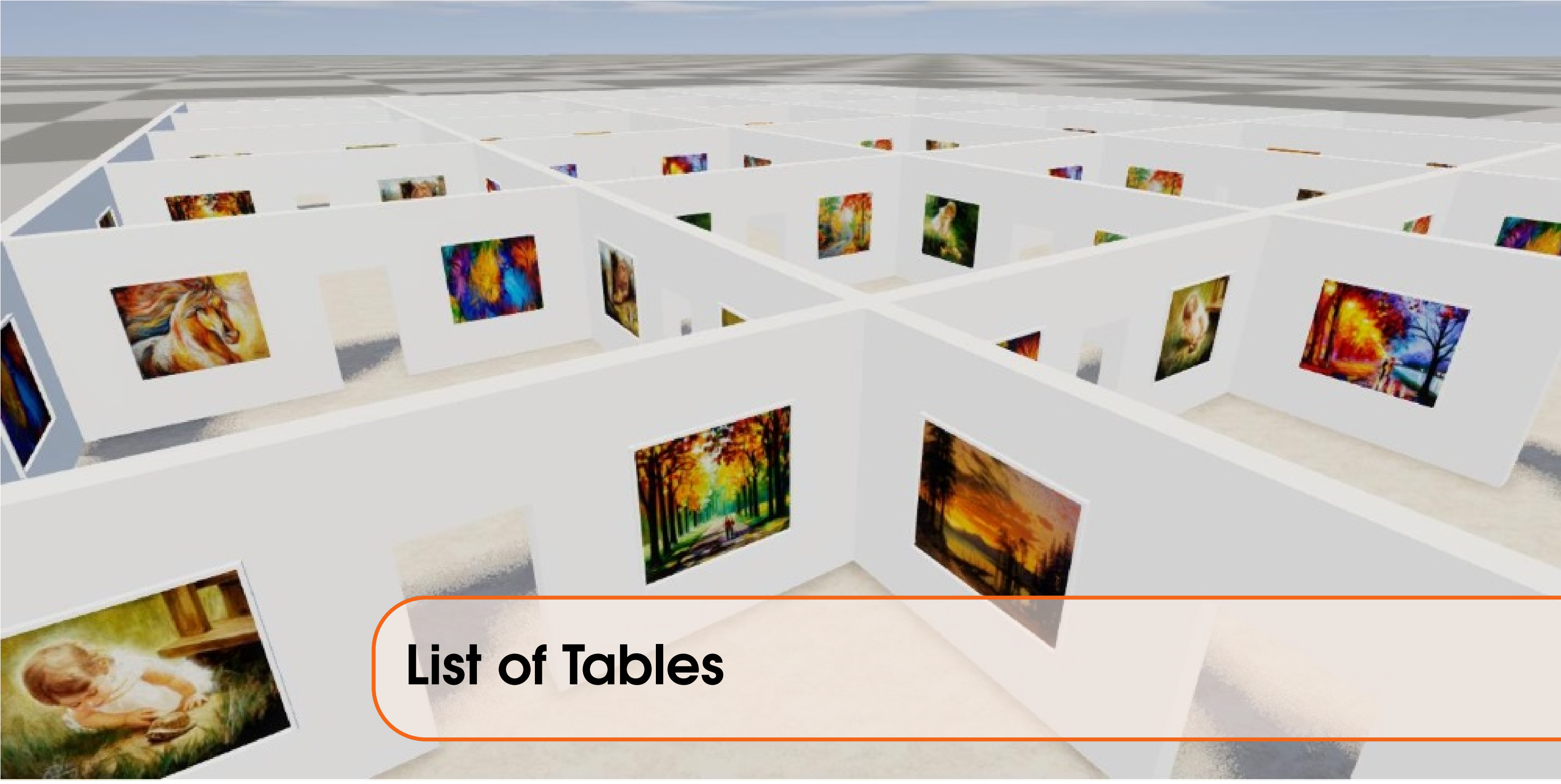
Finally, to all my friends, with a special attention to those who accompanied me in Portugal, Switzerland and the Netherlands in the last years, a huge thanks for all the joy, help and patience to endure my frustrations, for all the nice beer sessions, parties, duck lunches, francesinha dinners, movie sessions, board game nights, LAN parties, trips and many other activities that took place in these last years. A special thanks goes to my girlfriend, Joana, for taking the time and effort of proof-reading this long document, always with very sweet, constructive and helpful feedback.

Last, but definitively not least, my gratitude goes to my family, in particular to my parents and sister, for their promptitude, their advice and constant support, as well as their understanding and patience, especially in the hardest moments.



Contents

List of Figures



List of Tables

1. Introduction

Virtual urban environments are recurrent and popular in virtual reality applications since they can serve multiple activities such as games, movies, urban planning, training, simulation, education, and advertising, among others.

They are however, also among the most complex environments to produce and manage: they are composed by different types of features and entities, each with its specific nature, such as terrain, roads, vegetation, buildings, furniture that can be spread throughout large areas and comprise a high level of detail. The design of the various content types poses its own design challenges and their integration must obey certain rules and directives. Most structures are often unique and specific and require different building strategies.

Depending on the application, requirements may go far beyond the geometric representation. In games, environments may include quests locations, enemies, weapons spawn points; in simulations, they may include events, behaviors; in urban planning, detail may not be important, but rather the possibility to visualize data with certain visual aids and reporting information. In short, application-specific objects need to be introduced during the design process. Yet more than one solution is required to maximize the efficacy of the result, for instance, to provide diverse exercises in a training environment.

Due to these requirements and complexity, the production time and cost are often high. In the entertainment industry, this is somewhat compensated by the fact that the budget is high. However, for lower-budget companies and academic research, such resources are not available, restricting the possibilities of their usage.

The major impact of virtual urban environments in virtual reality applications, coupled with its high costs, high development time, and need for application-specific content, motivate the development of faster and more powerful development techniques that help in its production.

Procedural generation proved to be extremely useful in the generation of content in several virtual reality domains, showing strong prospects for application in the design of virtual urban environments.

1.1 Problem Statement

Procedural generation techniques are presented as a solution for content production to the extent that they are able to generate it almost automatically, requiring only the definition of rules, constraints or parameters to control the generation process.

This paradigm contrasts greatly with the concept of manual, interactive manipulation of content, where the user is given both the power and burden of specifying every detail of its creation. Although this allows for greater expressiveness, this approach does not scale: for building large virtual urban environments, it would require an immense amount of effort to focus on the creation of each individual element of the scene.

Procedural content generation has proven to be especially suitable in the generation of urban environments since these are packed with patterns and configurations that can be reproduced using computational algorithms. However, when it comes to reproducing specific designs, not all alternatives are equally expressive. When attempting to reduce the effort of the users, providing declarative and high-level means to generate content, often they end up restricting their descriptive power, which makes the reproduction of certain designs difficult, if not impossible to achieve. It is a problem of *expressive power*.

To address this issue, many approaches rely on building rules using lower-level grammars or text-based scripts to sequence the computer instructions to generate the content. Although they provide a finer design control, they are unattractive to designers and have a steep learning curve. They are often difficult to read, understand and manage, especially for large sets of rules and for more complex designs. It is a problem of *manageability*.

Another great challenge consists in properly accessing and controlling the large amounts of data that is produced and operated on. Many approaches rely on iteratively working on single objects, progressively evolving them into more complex ones. However, object development is often dependent of their context, so taking a look of an objects neighborhood or even at the whole set of available objects may be required. For instance, consider placing an advertising billboard on the unobstructed wall of the highest building, in a location that can be visible from all others in the block. This requires figuring out wall obstruction, determine the building with the greater height property and perform visibility tests on all remaining buildings that belong to the same block. The effective control of such content variety is again a problem of *expressive power* and *manageability*.

One should also consider that, for each type of feature (e.g. terrain, tree, road, building exteriors) and each type of style (e.g. roman, gothic, modern), many different methodologies exist, each being more appropriate to address specific details than others. The use of different data structures is common as each technique requires specific object semantics, i.e. a particular knowledge of the manipulated object for a more optimized generation and manipulation. It is also important to consider that, depending on the global purpose (e.g. building exterior, building interior, or both), different strategies may be required. A major problem lies, therefore, on how to control and manipulate different features, data structures and approaches within the same pipeline. It is an issue concerning *manageability* and *integration*.

When reproducing real-world urban environments, procedural content generation can benefit from the integration with external data sources that describe those spaces. Geographic information systems (GIS), photographs and textual descriptions are examples that carry a lot of information about the semantics of each environment object and therefore can be used to guide the automatic generation process. However, these sources tend to follow their own formats, structures and standards and are often unorganized, incoherent and incomplete. Addressing and integrating such format diversity implies that specific rules have to be designed to deal with the available data. Again, it is a problem of *manageability* and *integration*.

Virtual urban environments consist also of multiple types of content: models, textures, sound, but also objects with physics, lighting, animations, behaviors, etc. A great deal of research work has been invested in the sense of addressing the procedural generation of the geometry (a subfield commonly known as procedural modeling) or the textures that are applied to it. But instead of being produced in different domains, the processes could be merged. Street signs with corresponding street names could be generated after the streets have been placed or named store banners could be added after the houses have been built. Likewise, the generation of application-specific objects could accompany the geometry generation process: light sources can be instanced where the lamps have been placed; triggers can be placed and connected to other objects (for example light switches or elevator buttons); character spawning areas can be defined in no-exit streets; racing checkpoints can be placed according to traversing algorithms on the street network. In other words, the semantics of the generated geometry objects can be used to generate additional types of content within the same pipeline, producing more complete environments. The challenge lies, however, in how to organize and operate a scalable solution that handles all such content types effectively. It is an issue of *integration*.

Considering all these problems, one can state that more expressive and manageable solutions are necessary in order to control the procedural generation processes:

- They should provide high-level manipulation facilities, yet allow a more precise, low-level control, if needed. This would facilitate readability and control, yet retain the necessary expressiveness to build specific designs.

- They should consider all the needs for accessing and managing large sets of data, to operate on them through context-based methods and to introduce data from external sources;

- They should allow the integration and manipulation of diverse types of content within the same pipeline, so as to benefit from the content semantics at every step, as well as to produce more complete sceneries that include, for instance, application-specific objects.

Having this in mind, the main hypothesis to be proved in this Ph.D. thesis is the following:

The production of comprehensive virtual urban environments, integrating diverse types of content, can be better expressed and managed by a unified rule specification language to guide procedural content generation methodologies.

1.2 Research Questions

Having in mind the problems and hypothesis presented in the previous section, the research objectives are reflected in the following questions:

- **1.** How can expressiveness be improved to achieve a more broad and comprehensive range of designs?
- **2.** How should procedures and objects be represented, so as to benefit from their inherent semantics and improve content manageability?
- **3.** How can different types of content be generated and integrated within a single pipeline and what advantages does it encompass?

1.3 Contributions

The contributions of this work are as follows:

- Development of a visual, graph-based rule specification language that allows a powerful control over the content creation and transformation, extending the reach of design expressiveness and manageability - the procedural content graphs;
- Generation and integration of the several types of content that constitute a virtual urban environment, in a single generation pipeline, harnessing the descriptive power of their semantics and their context;
- Development of a framework and design system - the Construct - that allows a flexible introduction of new content types and algorithms, along with operators and manipulation tools that allow their visualization and analysis, as well as their incorporation within other systems.

As such, the contributions of this work can be described as a strong improvement of the following aspects:

Expressiveness (or expressive power), which refers to the breadth of structures and ideas that can be represented with this language. By providing the means to generate a broader range of structures, many design limitations present in grammar or other rule-based approaches have been addressed. This has been made possible through the generic node-based representation, where any imaginable operations and ideas over individual, sets or lists of entities can be controlled through configurable parameters and constraints. Its design paradigm focuses greatly on encapsulation of graphs into more complex operations, introducing better semantic control over the developed entities and promoting the work on several levels of abstraction. As a result, a wide range of techniques and strategies can be employed to achieve a greater control over the content generation.

Manageability, which refers to the ease to manage both the structure of the procedures and the flow of content data. On the one hand, the new graph-based language consists of a visual representation, which makes content flow easier to follow and to operate on. On the other hand, it allows the manipulation and combination of large amounts of data, introducing sorting, filtering, combination and aggregation operations, among others. Through the use of imperative-based instructions, such as reference passing and state persistence, the range of possible design strategies is enlarged. The use of encapsulation contributes to a better separation of concerns, reusability of large operation chains and the definition of execution scopes. This is complemented by the introduction of dynamic loading, which can make the generation process simpler, faster and easier to encode.

Integration, which refers not only to the possibility to combine and manipulate diverse types of content and feature types within the same pipeline, but also to connect to other systems with different data representations and purposes. The co-existence of such content variety means that complete scenarios can be produced in one step, while allowing different types of entities to interact and influence one another. The integration with external systems, on the other hand, is beneficial for obtaining data that affects the generation process, for exchanging command, analysis and report information or to export the produced content either directly or through intermediate formats to other systems.

1.4 Publications

In the scope of this thesis several related publications were written and submitted to scientific conferences or journals, most of them being related or result of a certain project, activity, phase or milestone, which are hereby enumerated and explained:

Procedural Content Graphs for Urban Modeling [Sil+15]: This paper introduces the concept of Procedural Content Graphs, its beneficial impact of expressiveness, manageability and integration of several types of content, as well as its materialization into the Construct System. Started in the scope of the 9-month working period at the Computer Graphics and Visualization Group of the Delft University of Technology, this paper had the special contribution of Prof. Elmar Eisemann. The content of this publication is contained in chapters 3, 4 and 5 of the current document.

Node-based Shape Grammar Representation and Editing [Sil+13]: This paper presents a solution that employs a node-based visual language to achieve the same generative power as shape grammars, but featuring a paradigm that is more attractive to designers, while the grammar rules are automatically generated. This work was developed as a result of a 6-month internship at the Esri Research & Development Center in Zurich, under special supervision by Dr. Pascal Müller. It represents an intermediate concept which ultimately led to the development of Procedural Content Graphs.

A Collaborative Environment for Urban Landscape Simulation [SCR12]: This paper presents a collaborative solution for large virtual environment recreation, where multiple users may contribute with additional data, to either improve the model fidelity or to preview the impact of certain urban changes. Started within the scope of the "Modeling and Simulation" subject of the Doctoral Program, this paper had the special contribution of Prof. Rosaldo Rossetti, and aimed to respond to the needs of the 3DWikiU project (see section 6.1).

An Urban Ontology To Generate Collaborative Virtual Environments For Municipal Planning And Management [Mar+12]: This paper describes a methodology to integrate multiple sources of real data from diverse municipal GIS in a unified data model, which is based on the CityGML specification. This model is mapped onto an urban ontology to be employed by procedural modeling methods, to produce three-dimensional models of the urban environments. The work on the specification was developed by the master student Tiago Martins and integrated with the system built and described in [SCR12], also in the scope of the 3DWikiU project (see section 6.1).

A Procedural Geometry Modeling API [Sil+12]: This paper was the result of a first conception of an API for geometric manipulation in procedural modeling. This specific approach enables a more powerful control over the geometric entities by performing global selections based on their attributes, aided by the definition of a topological structure featuring scope, spatial localization and semantic information. This work was motivated and first developed in the scope of the "Assisted Reconstruction of 3D Objects" subject of the Doctoral Program, under special supervision of Prof. Rui Rodrigues and Prof. Augusto de Sousa, and was applied for the generation of large virtual urban environments.

3DWikiU - 3D Wiki For Urban Environments [Mel+11]: This publication was the first description of the 3DWikiU project 6.1, where the goals, architecture and functionality of the system were introduced and the following steps of development were presented. The writing was led together with Miguel Melo, MSc under close supervision of the remaining members of the project.

A Procedural Modeling Grammar for Virtual Urban Environment Creation [SC11]: This paper presents a grammar for describing procedural modeling guidelines based on GIS data. The focus is the integration of real-world data source queries, as well as the provision of management facilities to handle their large number, dimension, format and level of detail. The grammar can then be integrated into a procedural modeling tool for generating very large virtual real-world urban environments. This work was a first attempt to develop and enhance a specification for procedural generation that would surpass existing ones.

Procedural Modeling for Realistic Virtual Worlds Development [SC10]: This paper presented PG3D System, the procedural modeling solution developed in the scope of the precedent Master thesis, a source of inspiration, experience and development for the current Ph.D. thesis approach. This system's implementation in spatial database management systems induced fast data access, large data manipulation features, complex query capabilities and format flexibility, allowing the creation of large scale virtual environments, imbued with optimized data structures, compatible with any digital game, custom modeling tool or metaverse platform.

1.5 Document Structure

This document is organized into 7 chapters. The first and current chapter presents an introduction, followed by the motivation and problem that instigated the current Ph.D. work and leads to the formulation of a hypothesis. The research questions and corresponding contributions are then introduced, followed by an enumeration and description of the publications written during the development of the Ph.D.

The second chapter presents the current state of the art, where the requirements and challenges of virtual urban environment production are clarified. Procedural generation techniques are addressed afterwards, first explaining the concept, then mentioning multiple examples of its application in the development of virtual urban environments. It concludes with a description of the unaddressed issues behind the control of such procedural methods.

The third chapter presents the main contribution, the Procedural Content Graphs. Here, the details of its visual representation will be explained, from the nature of nodes and ports, to the flow of entity data and how it can be controlled using aspects and augmentations, as well as how it can be organized using encapsulation. This will be complemented by the possibilities to enforce an imperative paradigm and to dynamically load nodes, both aiming at a more flexible graph and data management. It finalizes with an overview on how semantics can be defined and manipulated using such a graph methodology.

The fourth chapter focuses on a practical application of Procedural Content Graphs, addressing issues and providing task examples that would be very cumbersome or even impossible to express with previous approaches. This includes operations such as merging, aggregation, sorting, context awareness and constraint-based generation, among others, which contribute to an improvement in both expressiveness and manageability. The chapter finalizes with a detailed description on type integration advantages and how it can be used to produce more comprehensive environments.

The fifth chapter describes the implemented solution, the Construct system, composed by a framework that features several procedure libraries for content generation, and a visual interface, composed by many plug-ins focused on the creation, manipulation and organization of Procedural Content Graphs, as well as the visualization, debugging and export of the generated content. The chapter finalizes by presenting the possibility to connect and incorporate the implementation within external systems.

The sixth chapter performs a case study validation based on several academic and research projects which have embraced Procedural Content Graphs and the Construct in their solutions.

The seventh chapter is dedicated to the presentation of results and corresponding discussion. It will start with comparison with existing approaches and tools, followed by an analysis on the generation performance, proceeding afterwards to a discussion on obtained user feedback. The chapter finalizes with a discussion of a tutorial-based user study that was performed on the Construct.

The eighth and final chapter will answer the raised research questions, based on the presented work, after which some future work perspectives will be outlined.

2. State of the Art

The purpose of this chapter is to provide a better insight on the subject of this thesis, as well as analyze and understand the requirements and related works on virtual urban environments production, which can serve the field of virtual reality.

As a first step, the incidence of virtual urban environments in games will be studied, exposing some general challenges and processes in level design - which are inherently part of virtual urban environment design, while referring to some of the explained game examples.

In a second phase, procedural generation concepts will be presented, explaining their advantages and disadvantages, as well as examples of their successful application. This section serves as an introductory phase to the methodology, being its specific relevance in virtual urban environment generation presented in a last section of this chapter. There, a closer look on work regarding the expeditious production of the various urban environment issues - terrain, nature, street, building and other entities - will follow.

The chapter will conclude with a summary of the presented contents, while presenting some open issues in the development of virtual urban environments.

2.1 Development of Virtual Urban Environments

Developing three-dimensional virtual environments embraces a set of challenges for their producers, which are transversal to all applications that use them. On the other hand, there are many issues, derived from its gameplay and objective-oriented nature, which are part of the virtual world and therefore have to be considered in the creation process. Urban environments, being a subset of such environments, inherit these same properties.

This section presents therefore some of these challenges and processes that should be subject of attention in virtual urban environment development.

2.1.1 Challenges

When introduced in the context of gaming, the concept of virtual environment requires a superior set of processes in order to fulfill its role within the application. Whereas software such as Google Earth [Goo10] has a more informative rather than entertaining or educational purpose, virtual environments in games have additional requirements, which labels them as "games". As seen in the two previous sections, such spaces are used to different extents and scopes, which make their specific requirements harder to generalize. However, some studies [Ale+05; SJ04], presented some broad-spectrum elements that players consider essential in a three-dimensional virtual environment and that may contribute to higher levels of knowledge, learning and skill transfer.

- **Immersion and Suspension of Disbelief:** Immersion refers to the degree to which an individual feels absorbed by or engrossed in a particular experience [WS94]. One of the elements that players consider that increases their sense of immersion is the existence of detailed graphics and sound, fitting the intended atmosphere. Just as sound can heighten suspense in a horror movie, it is important that issues such as lighting and graphic textures have the same effect. Likewise, it is necessary that, once established a certain theme, the virtual environment of the game is adjusted. As denoted by [Ale+05], immersion is therefore based on the extent to which the visual displays support an illusion of reality that shuts out the physical one, either through range of sensory modalities or through the game story, tasks or atmosphere - as well as how they match together.

- **Presence:** Also referred to as situated immersion, presence refers to the subjective experience of actually existing within the computer-mediated environment even when one is physically situated in another [SS00]. This is related to the amount of human-computer and avatar-avatar interaction required by many simulations, which can be supported by a greater degree of control, environmental richness, scene realism and physical environment modifiability.

- **Interactivity:** One of the much criticized by players is that their interactions with the environment are very limited, hurting therefore the sense of presence. In most cases, the objects presented in virtual environments are part of the scenario and cannot be moved or affected. The furniture is not removable, the walls cannot be damaged and the papers cannot be read. There have been only a few titles that have implemented more advanced levels of interactivity, but the results have been very positive.

- **Fidelity:** Fidelity can be described as the extent to which the virtual environment emulates the real world. In this sense, [Ale+05] presents three categories: physical, functional and psychological fidelity. The first is defined as how the physical simulation looks, sounds and feels like the real environment concerning visual display, controls, audio and physics models. Functional fidelity is related to how the simulation reacts to the interacting equipment. Finally, psychological fidelity is linked to the replication of psychological factors (e.g. stress, fear), which are experience in the real-world. When designing virtual environments, physical fidelity is the most controllable aspect, but the others are inherently linked and therefore must be taken in consideration.

- **Consistency:** One of the most important aspects is the perception that objects of the same nature act the same way. An example of non-compliance with this rule will be "one glass that breaks, while others don't." These inconsistencies can cause difficulties for the player to learn the rules of a game, which appears to be changing. It is thus important that, when an object has a different application, it shows specific features that induce a different type of interaction.

When considering training applications, it is important to consider the user acceptance or "buy-in", which refer to the degree to which a person recognizes that an experience or event is useful for learning [Ale+05]. Higher level of buy-in means that the user will invest more effort in extracting lessons from training, and even more to transfer such lesson to the real world. This effect is also produced from the previous elements.

Compliance with these and other needs in virtual environments is a difficult task to achieve. This is especially true when large teams, whose individuals may have different ideas and styles, are involved. Some of these aspects, such as consistency and interactivity, can actually be more easily achieved through procedural methods, since they act based on a defined set of rules and can quickly recreate a particular atmosphere, and from real-world data sources, in a virtual environment.

2.1.2 Processes

An important fact to consider in virtual environment design, and especially of urban nature, is that they consist of far more than purely geometric elements that form up buildings and terrain to navigate on. Level design is both an artistic and technical process that regards multiple issues, all of which have an impact on gameplay, and especially on the various aspects explained in the previous section. This section intends to give a brief overview of many of the processes, entities and choices that must be considered in level design [FS05; Gal11]:

- **Storytelling level design:** Environments have a purpose, a reason to exist, which matches the intended story and gameplay. The resulting environment must reflect the background story, without the need to explain it.

- **Flow of the environment:** Navigation within the environment must be planned, considering player starting points (also called "spawn points"); objective locations; danger locations; area accessibility (the areas where the is player allowed to go); level transitions and portals, etc. An important issue to consider is the map boundaries, and how to define them in a natural way. Creating unreachable, terrain elevations, street blockades, hazard barriers (such as fire) are examples of how to delimit the playable grounds without ruining the sense of immersion.

- **Illusion of Freedom:** By creating a more open-world scenery, it is possible to give the player an impression of freedom, while still guiding them to their next objective. The existence of multiple paths towards the same goals removes the sense of linearity and encourages the player to find multiple solutions.

- **Guidance:** The player navigation can be eased when guidance hints are provided. They can be implemented through straight ways, such as Head-Up Display (HUD) arrows or maps, or through more subtle hints, such as props, barriers or moving non-player characters.

- **Interactive environments:** Believability can be increased by having players interact with their surroundings. Non-player characters, doors, tools, and props are examples of entities that make a scene look livelier, especially when they can be used, moved or be subject to physics. Their positioning must also follow certain rules, otherwise the game may become too difficult or too easy when, for example, pickup items are found unequally scattered along a path.

- **Props Placement:** Whether interactive or not, props make the difference in the richness of a virtual environment, and the more detailed, the better. While many may exist with a purely decorative purpose, such as garbage, debris, decay, flies and immovable objects on the ground, other may serve, for instance, as player obstacles or cover points, therefore directly affecting the gameplay experience.

- **Character placement:** Characters, human or animal, are often very important in goal achievement. If they are not the goal themselves (for example, if they have to be rescued or killed), they can serve as aids or obstacles. They location (and orientation) in the level is therefore important to serve the game purpose. There is no sense in having guards facing the walls or firemen locked in closets if they are supposed to hinder or help the player, respectively.

- **Imperfection:** As most things in the world are not perfect, so should be the ones in a virtual environment. In urban environments, few are the buildings without fractures, rust or dirt, therefore such characteristics must be present as well. Things must be broken and made dirty because, likewise, items are often moved, broken, replaced and fixed up.

- **Scripted events:** There are certain dynamic events that affect the environment, forcing characters to review their choice alternatives. For instance, in a firefighting scenario, an explosion can block an exit, forcing the player to look for another one. While this is normally achieved by pre-defining the location of such events, ideally this should happen randomly, in order not to bias the player decision into a specific one.

- **Environmental danger:** There are some dangers that can be derived even from a static environment, such as fire; radiation; height (causing the player to fall); traffic; weather and smoke (causing low visibility); water; explosions. Solutions for circumventing them should also be made available, either by going around (though destructible walls, ladders, air vents, etc.) or by providing tools that can fight these hazard situations (e.g. fire extinguishers and special suits).

- **Lighting, Color and Theme:** Levels should by nature follow a consistent style, or theme, which is important to guarantee consistency. Light and color are vital to describe spaces, and their positioning is not trivial. In open urban environments, the sun is the main light source during the day and is replaced by the moon and street lighting at night. They build up an atmosphere, which can be reinforced by additional visual effects, such as fog, haze, smoke, fire, and weather system.

- **Sound and Music:** The audio component of an environment is of extreme importance to promote immersion within the game. Sound should be present in character movement, speech, hits, fire, water, and others. Every action, every environmental occurrence should be accompanied by a timely, matching sound. For instance, if a building is on fire and there is no complementary sound, it loses its credibility. Music, although not as essential, contributes to the atmosphere of the game, in a similar way that it does in movies, having effects on psychological fidelity, stimulating feelings such as stress or fear.

2.1.3 The Role of Semantics

As shown in the previous section, every element in a level environment has its own specific purpose in the scope of the game's objectives, gameplay and consequent impact on the player. In most cases, this is achieved by instancing a set of predefined, game-specific class entities, on the level terrain. Each instance is manipulated manually in terms of position, visual/geometrical representation or other properties. Their combination in the game level is then used to achieve the desired effect. Although it works, this approach can become very complex to create and manage when editing huge worlds. By manually placing single (mostly isolated) game objects, it is hard for a level designer to fulfill his original intent, then maintain it in subsequent modifications, or even reuse elements in future designs. This may also cause more serious problems related to level consistency, if the same design ends up having different outcomes, or if objects with the same apparent functionality behave differently in the same situation.

The main problem behind this approach lies on the lack of object semantics, i.e. a meaning, as expressed in natural or symbolic language. By mapping game objects to real concepts, it is possible to imbue them with a considerable amount of knowledge about themselves, about possible behaviors or relationships with others, as well as their importance within the game world. Objects may have properties other than geometrical and physical attributes. They may act in accordance to other objects, providing services and deeds that can be affected by world events (this has been achieved in [KTB09]). An example of such an object would be a plant seed: it can only be placed on appropriate terrain; it evolves into a fully-grown plant according to the season and weather, which then provides nutrition to animals and people. A semantically rich world provides all these concepts – the vocabulary – to define such broad and realistic interactions [Tut+08].

By using such a semantic definition it is easier to achieve the intended consistency, reusability and maintainability in game design. If the concept is declared once, it can be used over and over. If the relationships are carefully planned, the objects will behave among themselves always in the same way. And lastly, if the defined concepts are comprehensive enough, they can be used to capture the designer's intent by just declaring, at a high-level, the expected outcome.

In [Tut+08], three levels of semantic specification are distinguished:

- **Object semantics:** This most basic level contains physical and functional properties that are specific to that object. Physical properties include material, dimensions, and mass. Their definition can be somewhat automated, such as stated in [InDM06] (e.g. the dimensions can be obtained from the model geometry). Functional properties, on the other hand, may include interaction possibilities as well as details on how they can influence others. This has been already introduced in the concept of smart objects [KT98].

- **Object relationships:** Relationships can exist between instances of objects (for example, if they are spatially close to one another) or between object classes. Different objects may share some properties, both on a physical and functional scope. An important relationship type is inheritance, which can be expressed in a taxonomy [Bit02] to classify and group classes of objects sharing similar properties. Other kinds of relationships may include ownership, causality, aggregation and inclusion, which are association types commonly found in ontologies.

- **World semantics:** On a highest level of specification lie the world semantics, which are defined on a more global scope that affects all the game environment entities. Daylight, weather and seasons are aspects that influence ecosystems, plant and animal distribution, as well as their behavior. Contextual information, such as demographic, economic or even criminal traits can have a global impact on the game development. This is especially important in business and urban simulation games.

2.2 Procedural Generation in Games

Procedural generation is a term which is becoming increasingly popular and recurrent in various areas of computer graphics and especially in the area of virtual reality and gaming. In this section the concept will be explained in more detail, considering benefits, disadvantages and common strategies employed in the various areas it encompasses. Examples of procedural generation application will also be presented, describing the reason and outcome of its use.

2.2.1 Concept

Procedural generation is a commonly used term to describe a methodology that seeks to produce automatically, rather than manually, any kind of media content (models, textures, sounds, music, objects, etc.) through a set of techniques, computer algorithms and a certain degree of randomness.

The first approaches on procedural generation began with the mathematician Benoît Mandelbrot, when he introduced the term "fractal" in his work [Man83] to describe a family of shapes, which, when following some pattern repeatedly, could result in more complex shapes. The process is based on an iterative and recursive approach of a defined equation. The final generated results indicate a strong resemblance between each component to the whole (see Figure 2.1). This fact constitutes one of the basics of procedural generation.

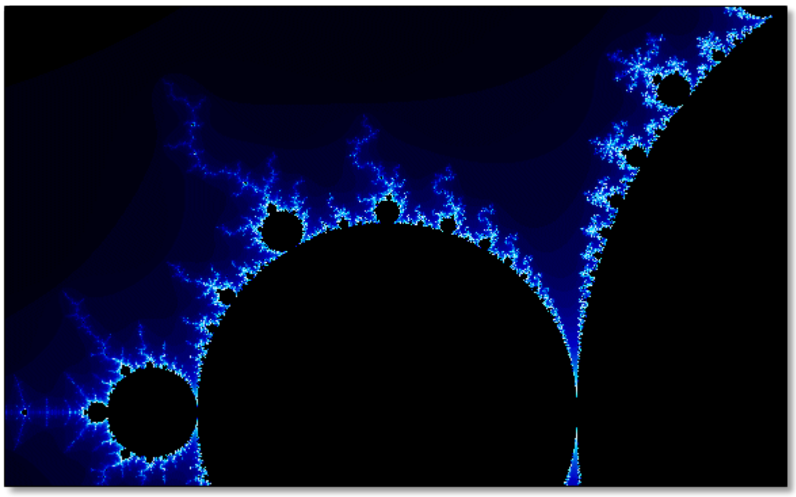


Figure 2.1: Zoom on the Mandelbot Set [Loy02].

Some definitions label procedural generation as "totally random" and "nondeterministic" [You09a]. While it is true that procedural generation can produce a multitude of different results with the definition of random data (within the computational limitations of generating such randomness), it is not absolutely necessary. It is trivial to make the computer generate the same sequence of random numbers, thus producing the same "random" content each time it is viewed. Thus, such content generation can be deterministic, being the result automatically and randomly produced by the computer always in the same way, if desired [You09b].

The scope of application of procedural generation constitutes a source of great divergence of opinion. When considering its application in virtual reality applications, such as games (which is the focus of this analysis), its meaning is far more specific. The concept has been given multiple definitions which, although not contradictory, indicate slightly different approaches [Hal08; Int07; PCG09a; PCG09b; Tog+10; You09b].

The most common view on procedural generation concerns its opportunity for rapid development, being therefore an essential tool for producing large amounts of content in lesser time, requiring much smaller work teams and therefore resulting in much lower production costs [Wat+08]. Generally speaking, this view would end up being transparent to the final consumer, who would be able to perceive those contents in the exact same way.

Other definitions refer to procedural generation as a way to create media on the fly, rather than prior to distribution, requiring therefore less storage space, but generating the same results every time the content is viewed. On the same basis, some definitions make a distinction between "procedural generation" and "procedural content generation" [Dou08; PCG09a; PCG09b], being the main difference the fact that procedural content generation actually produces different results each time it is run, thus affecting the experience on each generation. In [Tog+10], no such difference is pointed out, however, game content is considered to be everything of a game that affect gameplay, but excluding non-player character (NPC) behaviors as well as the game engine, which, by such definition, excludes NPC artificial intelligence. Listed types of in-game creatable data are "terrain, maps, levels, stories, dialogue, quests, characters, rulesets, camera viewpoint, dynamics and weapons" [Tog+10].

2.2.2 Benefits versus Manual Creation

Perry and Roden [RP04] point out four drawbacks of manual content creation.

Firstly, the fact that technology has evolved, providing both production and visualization tools, has contributed to an increase of the expectations of the end-users towards the media contentment, which ultimately means that artists need more time for their creation. Secondly, handcrafted content is harder to alter after it has been created. Since specification or design changes can also end up in technical requirements for content, these may end up becoming obsolete (especially considering formats, tools, etc.). Thirdly, there is the common need for content designers to resort to external conversion utilities to translate their created media from the authoring tools to the visualization/gaming engines. In that process, incompatibilities and inconsistencies between what was created and what is ultimately viewed in the engine are very likely to occur, requiring successive trial and error tests. Lastly, Perry and Roden state that interactive applications, such as games, have the potential to become much more expansive: there will be the time when no longer humans can keep up for its creation.

According to [Gu6; Hal08], procedural generation comprises the following benefits, among others:

- **Reduced labor costs and increased productivity:** For producing certain contents, procedurally assisted systems require less interaction / input, resulting in less work to the designers responsible for the task, thus increasing their productivity.
- **Data Compression:** If data can be generated on-the-fly, then there is no need to store it on a medium or send it over a network. Instead, using only generation seeds or guidelines may considerably reduce overall data size, at a cost of some more CPU usage to perform the generation tasks. The possibility to accurately reproduce information on the receiving end from a small set of data decreases the need to store redundant information [Tog+10].
- **Generation of content by users:** Not always do players have the time and perseverance required to produce quality content for games. However, when procedural systems are involved, where only a few parameter entries are needed, this becomes simple and affordable for anyone. A classic example is the character configuration system in Role Playing Games.
- **Generation of content by programmers:** Even teams lacking designers can afford the luxury of producing high quality content and size by programming procedural generation methods that produce an infinite amount of alternatives through a small set of random entries.

2.2.3 Disadvantages

Despite the listed advantages, it is not always easy, appropriate or even possible to apply procedural methods for content generation. On the other hand, it is also possible that the use of such processes results in other sorts of difficulties:

- **Complexity of Implementation:** While it may result in reducing the level of staff resources dedicated to the creation of a variety of content, the implementation of procedural generation methods typically requires more qualified manpower, possessing both knowledge in design and programming [Dan07; Rem08]. Many existing procedural generation tools may reduce this problem, but, on the other hand, may not always match all the intended generation requirements.
- **Repeatability and lack of creativity:** Being created from a limited set of rules and a certain degree of randomness, procedurally generated content has the risk of becoming repetitive and unimaginative over time. If not carefully programmed, the same content can be generated in close proximity, making the whole set unattractive [Dan07].
- **Longer waiting times:** When used at the time of execution of the game, the procedural

generation methods, which by default are more CPU-intensive, may result in longer waiting periods on less powerful machines, which is something not any player is willing to tolerate [Int07].

- **Social Factors:** In general the procedural generation methods fail to produce human-oriented content. Dialogues, voices, and narrative descriptions are a few examples which are hard to generate automatically [Dan07].

- **Content Validation and Testing:** The introduction of procedural generation hence allows easy creation of large-scale content (such as environments) through a reduced set of inputs and generation rules. However, the introduction of a new rule may have less obvious repercussions in other ones. On the other hand, the verification of this fact, given the large size of the product generated, is also more complicated [Rem08]. In this sense, more suitable testing methods are needed.

2.2.4 Teleological vs. Ontogenetic

Procedural generation is often used in the reproduction of existing real-world features such as terrain elevation, natural organisms or even man-made structures. The correct understanding of how these processes actually take place, how they are organized and how they influence each other is one of the basis of procedural generation techniques and many popular algorithms are actually derived from such observations.

According to [Gu6; MP06; Wes08], there are two main methods for procedural content generation: teleological and ontogenetic.

The teleological approach, created by Alan H. Barr, is a technique which seeks to reproduce a physical model of an environment, simulating a process which ends up generating a certain result. This method is mainly used in scientific applications where the process is important, and the end result must be as accurate as possible. Due to the very complex nature of such models, the procedure is very CPU and graphics intensive, being generally not recommended for real-time graphics applications such as games.

The ontogenetic approach, introduced by F. Kenton Musgrave, is, on the other hand, more focused on the final result, regardless of the necessary process. This is achieved by first observing the end results and afterwards trying to reproduce them through more coarse approaches. This technique is similar to the theory of Benoît Mandelbrot to describe a complex object as a result of its fractals, and is commonly used in real-time applications such as games. In algorithmic terms, teleology and ontogeny in procedural generation can be seen, respectively, as top-down and bottom-up approaches. The teleological (or top-down) strategy starts at a more general view of the system, and going deeper at the time. The ontogenetic (or bottom-up), start by focusing reproducing the details first, piecing them together to give rise to the greater system.

It is important to consider these two approaches depending on the situation, aimed object and target application for procedural generation.

2.2.5 Semantic Generation

The role and importance of semantics has already been mentioned in section 2.1.3.3. When such semantic data is linked to procedural generation techniques, the potential, quality, consistency and realism of these techniques can be improved. When dealing with semantically rich objects, it is much easier to specify constraints on how they are expected to look, behave, interact and relate to other objects. On the other hand, procedural methods can be useful in propagating the semantic information to the generated objects.

These advantages have been explored in the works of Tutenel [Tut+08; Tut+09], which consisted in a constraint-based layout solver, acting based on object relationships and behaviors. The three levels of semantic specification mentioned in section 2.1.3.3. are expressed with the help of constraints, which are rules on one or more variables, typically corresponding to object properties. In [Tut+08], the following constraint types are enumerated:

- Numerical: based on algebraic expressions;
- Geometric: based on geometrical concepts such as distance and dimensions;
- Physical: such as gravity or temperature;
- Time: to address object properties that may change over time;
- Distribution or variation: indicating how objects should be distributed, including their absolute numbers or ratios.

The presented solving approach in [Tut+08; Tut+09] is integrated with a semantic class library, which has a hierarchic setup, i.e. classes inherit properties, placement rules and features of their parent classes. These features are 3D shapes containing tags, used to derive valid and invalid positions by defining overlap rules. For instance, the offlimit feature indicates solid areas that do not allow overlapping, while clearance indicates areas that should remain free for object interaction (such as the area in front of doors). A layout planner submits objects to the solver one by one, according to the rule order that must be executed. The planner is also used as a way to manage manual user placements. The layout solver is then responsible for making sure the rules for each object are applied (see Figure 2.2). Since multiple solutions can be generated, layout results are classified according to a certain selectable criteria, such as the remaining amount of open spaces, or number of storage locations.

This work shows how semantics can ease and empower the procedural processes. Similar high-level semantic approaches, focused on a more declarative approach on procedural modeling

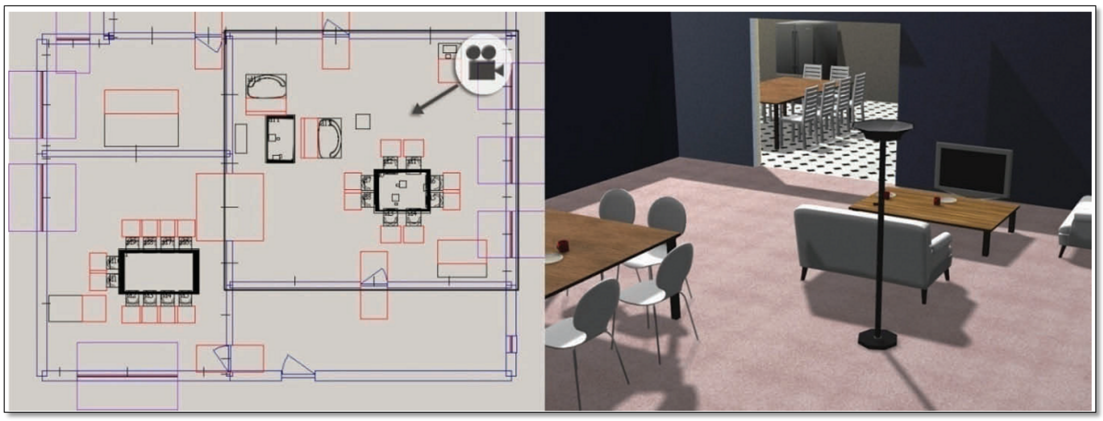


Figure 2.2: Automated generation of a house using the layout solving approach [Tut+09].

of large virtual environments, have been presented in [Sme+11], as it will be shown further ahead.

2.2.6 Application Examples

The concept of procedural generation is not new in the gaming domain. One of the first use cases have emerged in the 80's, called Elite [Aco88], a spacecraft simulator, which offered eight galaxies, each containing 256 planets and their space stations, all waiting to be explored, all this being procedurally generated [Joh09].

Many other titles have emerged over time. The highly acclaimed Diablo [Bli12]; The Elder Scrolls saga games [Bet14], such as Oblivion and Skyrim (see Figure 2.3); or the Hellgate: London [Han11] are just some examples of games that provided very large environments, all procedurally generated [Dou08]. Yet other games have used such methods for the easy assembly of characters, including the game Spore [Tot09]. Recently, the game Borderlands [Tak09] has shown to be able to provide more than half a million different weapons, which are procedurally generated on runtime [Bor08].

Figure 2.3: .Kkrieger [.th04] (left) and The Elder Scrolls: Skyrim [Bet14](right).

Over time, other titles use procedural methods to assist in development, but as in the examples mentioned above, the majority focused its application in only one area.

Recently, there have been deeper attempts to implement procedural means in almost all aspects of development. An example is the game .Kkrieger [.th04] (see Figure 2.3), a First Person Shooter, in which all content, including textures, models, animations and sound are generated at the beginning of the program, thus compressing the entire game into a single 100 Kilobyte large executable [Ben09]. This constitutes thus one of the more far-reaching approaches on procedural generation until now, and demonstrates the extent to which procedural methods can be used in games.

2.3 Procedural Generation of Virtual Environments

The development of virtual urban environments has always been a challenge for its creators. This can be explained mainly by the need for modeling a large number of buildings and structures, which should have distinct characteristics, varied and preferably detailed in order to avoid monotony and to provide some uniqueness to each section, as well as a greater realism and immersion in the environments.

On the other hand, urban settings usually follow a set of architectural patterns which can be replicated and, when properly combined, can produce a great variety of convincing results. This can be very well performed by procedural methods, which act based on sets of rules and, if intended, on geographical data.

In this section various processes of procedural generation of virtual urban environments will be presented, regarding terrain, plants, streets, buildings and other types of urban content.

2.3.1 Urban Data Sources

Modeling virtual urban environments requires knowledge of their natural structure, planning rules and common organization patterns. A close observation of the reality, together with the employ of more methodically gathered data sources constitute therefore obvious procedures that must be executed when such environments are to be reproduced.

However, their formats, methods of acquisition and levels of application have varied among research works in this area, depending on the desired level of detail and visual fidelity when comparing with the original environments.

Many authors in this area [Fin08; M+06; M+07; Won+03] have based their approach on the definition of an integrated set of rules in formal grammars that describe common buildings elements (windows, doors, columns, pillars, roofs, arches, walls, ornaments, etc.) and how they are related. The construction of these rules has been mainly done manually by observing buildings characteristics and common layouts, by seeking books in the area of architecture [Wat+08] or, more recently, by analyzing photographs of building facades [M+07]. These approaches have produced high-quality results, enabling the rapid generation of large and varied virtual urban areas. However, in situations where exact real-world buildings are to be matched with, these sources are not always sufficient.

In the work of Parish and Müller [PM01] city models were recreated based on some of their characteristics, such as economical and demographic distribution, which fairly contribute to the spatial street and building layout. Such maps were enough to obtain approximate and realistic results, but certainly not exact.

On the other hand, many other authors [BN08; Coe+07; HYN03; SMS06] have tried to employ more precise data sources in the creation of the virtual environments, with the intention to increase their level of fidelity, such as:

- Global information gathered through aerial photographs and bidimensional maps (terrain, road, etc.);
- Local geometric information of buildings and facades captured through laser scanning devices or digital cameras;
- Digital Elevation Models (DEM);
- Geographic Information Systems;
- Other urban databases.

Facing the great diversity of such data sources, a problem arises concerning how to automatically interpret and join this data, which may follow different standards or even possess their own formats, incompatible with each other. Concerning this fact, there have been efforts in conceiving an unified model of urban environments, mapped onto an urban ontology, which can allow multiple data sources to be combined [MCS11; Mar+12; Mel+11]. Different types of objects (terrains, buildings, trees) can afterwards be joined through their geospatial properties, namely their absolute or relative position in the planet (georeferenced), as mentioned in [Coe+07; SC10].

An important fact to keep in mind is that not always these sources are enough to describe an urban setting, which motivates the introduction of amplification techniques, such as the addition of some randomness in model parts where information is scarce and the level of detail is not essential [Coe+07].

2.3.2 Terrain

In any three-dimensional virtual environment one of the basics consists in the creation of a surface where all objects are sustained and are able to move. In a closed building, such will be each story's floor, an essentially planar and spatially limited surface, typically holding a simple and constant texture. However, in exterior environments, it's quite common to observe surface containing variations along its extension, such as multiple types of terrain and distinct elevation levels.

The procedural generation of irregular elevation terrains has already been extensively explored [Dis09; KM06; Sme+09], with many interesting results from research and commercial products. Most of these approaches focus on dealing with height maps - two-dimensional grids of elevation values, i.e. for each point in a base XY-plane, one height value may be associated.

[Hna+10; Pey+09; Zho+07] identify three main approaches to generate terrains: fractal landscape modeling, physical erosion simulation and terrain synthesis from images or sample terrain.

Fractal-based modeling was introduced in the pioneering work of Mandelbrot [Man83] and since then a great variety of stochastic subdivision techniques have arisen. Fournier et al. [FFC82] introduced the popular random midpoint displacement technique to create fractal surfaces, being further developed by Voss [Vos85]. Miller [Mil86] described several variants of this method: the main idea consists in starting with a rectangle, where all the corners are assigned an initial height value, and in iteratively adding a new central point, whose elevation is calculated through the average of its corners in a triangle, square or diamond shape, and adding a random offset afterwards (see Figure 2.4). The offset's range decreases on each iteration, based on a parameter that defines the intended roughness of the terrain.

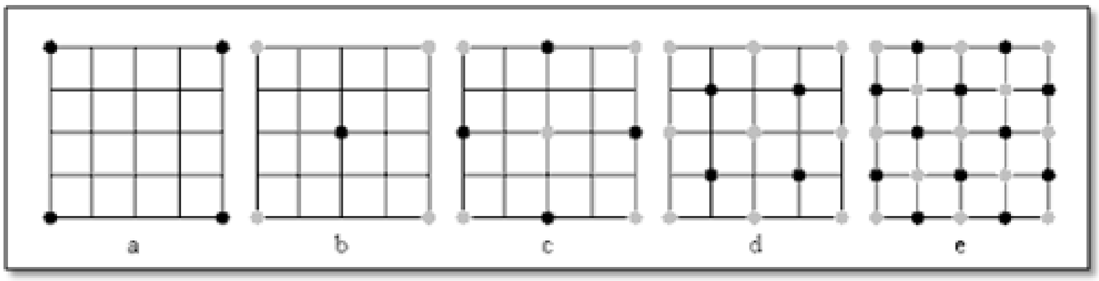


Figure 2.4: Mid-point displacement method for generating height-maps [Mar96].

Another very commonly employed fractal-based generation method is Perlin noise, initially developed by Ken Perlin [Per85], which consists in first creating a set of random values, which are then interpolated into coherent noise. It is possible to compose several layers of noise together to create more natural looking terrains [Sme+09].

According to [Zho+07], both these techniques are heavily employed in commercial software, such as Terragen^{TM} [Pla09], a popular tool for the generation, rendering and animation of realistic natural environments (see Figure 2.5). Developed by Planetside Software, Terragen^{TM} has been used to create photorealistic visual effects for many films and advertising instances. The software features control over "weather, landscape, rivers, lakes and oceans, suns, moons and stars" [Pla09], allowing shader customization, external data import, such as georeferenced terrain datasets as well as export in multiple formats. Other similar commercial products include

Vue [sof11], World Machine [Sch10], MojoWorld [Pan04], among others.

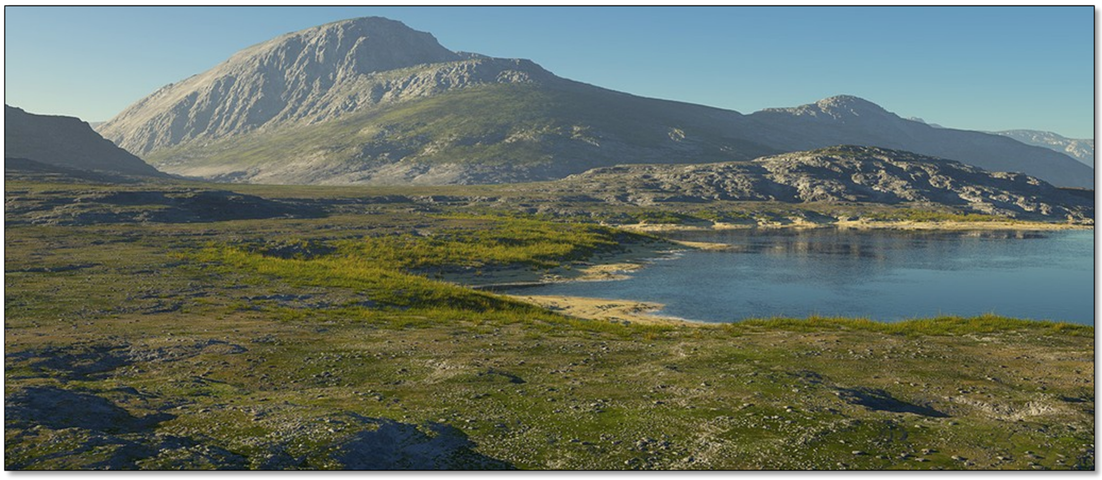


Figure 2.5: Planetside's Terragen^{TM} [Pla09].

The main downside of such commercial applications lies on their dependency on the main tool, therefore discarding the possibility to direct integration with real-time applications. [Zho+07] mentions that, in all these systems, synthesis occurs by changing global parameters, and therefore "do not support user user-specified placement of major terrain features".

Taking advantage of recent advances in graphics hardware, [SBW06] introduced these same techniques as through procedural shaders for multifractals, using GPU methods for real-time editing and rendering of procedural terrains.

Physical erosion simulation is an alternative approach to synthesizing terrains based on the reproduction of landscape formation and stream erosion phenomena. It is often used as a refinement step after generating a rough height-map. Thermal erosion diminishes sharp changes in elevation distributing material from higher to lower points, until a certain stability is reached; while fluvial erosion (caused by rain) forces material transport based on the local slope of the terrain surface [Sme+09]. Pioneering works in these field include [KMN88] and [MKM89], having the last combined fractal and erosion simulation approaches into a single framework. By having to run for many iterations, these algorithms are notorious for being computationally intensive, despite the positive and realistic visual impact of their results. More recent approaches, such as Neidhold [NWD05] and Olsen [Ols04] have aimed at real-time applications, such as computer games, having developed speed optimizations, delivering more reduced, but still acceptable quality results (see Figure 2.6).

Both fractal-based procedural and erosion techniques are guided by complex parameter tuning to obtain specific terrains. The third main approach consists, on the other hand, on constraint-based techniques which act based on user sketches and sample terrain. In [Zho+07], patches from sample height maps are used to generate new terrain. The synthesis is then guided by user feature sketches that specify their location in the synthetic terrain. A system in [GMSe09] is presented, which enables the user to draw the silhouette, spine and bounding curves of hills,

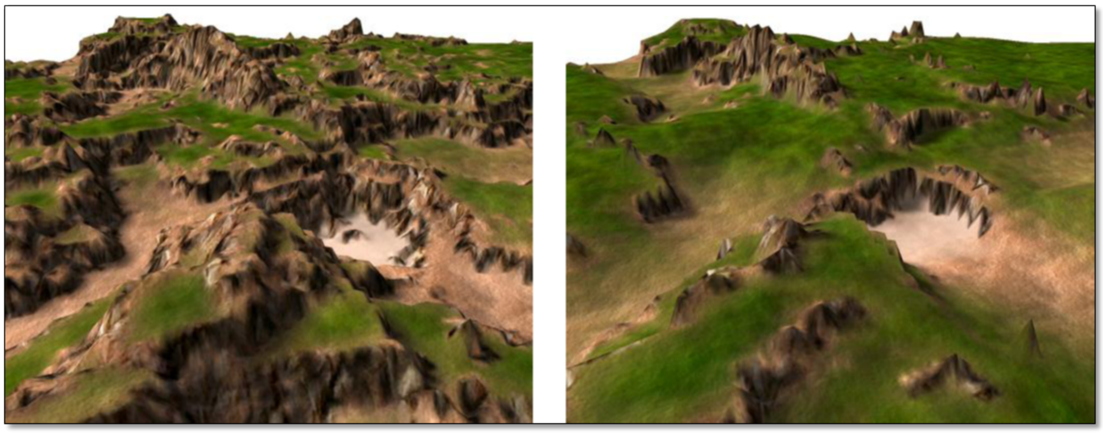


Figure 2.6: Application of thermal (left) and hydraulic erosion (right) [Ols04].

mountains, river courses and canyons.

The work in [Hna+10] describes a multigrid diffusion method for generating terrains based on sets of easy and intuitive editable feature curves, which handle the definition of ridge lines, river beds, lake and sea shores, and consequently hills and valleys. The generation is executed very efficiently, allowing real-time sketching and interaction with the conceived tool.

A limitation presented in [Pey+09] concerning the previously mentioned approaches lies in the fact that they rely on height field representation of the terrain, meaning that for each point in a base XY-plane, only one height value may be associated. In their work [Pey+09], the authors describe a technique which allows the representation of complex terrains with overhangs, arches and caves. Elements such as piles of rocks and pebbles are also simulated a tool which stabilizes layers of sand and rocks according to their repose angle (see Figure 2.7). All these features are integrated with a set of terrain authoring tools for greater user control.

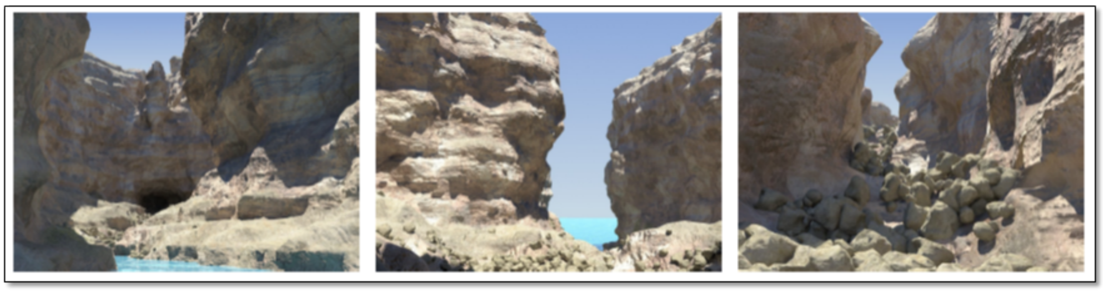


Figure 2.7: Canyon with detached rocks [Pey+09].

When the reproduction of existing terrains is intended, is it preferable to resort to real information such photographs or even geographical information systems, which normally contain terrain surface information in digital elevation models (DEM) [Har09]. However, since this information is not always available, generating approximate synthetic terrains may constitute a viable alternative, especially when user knowledge about its features may serve as input as well. On the other hand, even when only coarse or incomplete DEM information is available, employing some of the mentioned approaches can help improving existing data. This possibility has been successfully tested in [BA05].

2.3.3 Plants and Ecosystems

Natural landscape is a common component in every urban environment. Gardens, parks, sidewalk plants are some examples of their recurring incidence. Their manual creation, however, as well as their realistic placement and organization on a surface terrain constitute also a laborious task, but have been shown to be very efficiently executed by procedural methods. Fractal-like shapes such as trees or ferns are examples of the application of such techniques through relatively simple recursive algorithms, as proven in [Bar00], where a large number of such examples is shown. These approaches provide an abstraction from the structural complexity of the represented natural objects, using recursion to provide multiple levels of detail. They allow the generation of complex models from the recursive application of simple equations, working as a data amplification technique. Due to the self-similar nature of all fractal-based objects, these methods are however not effective for all such types of plants, since they may not contain such self-similar properties [KM06]. An attempt to alleviate such drawback is presented in [Opp86], where the introduction of random variables are proposed (see Figure 2.8) .

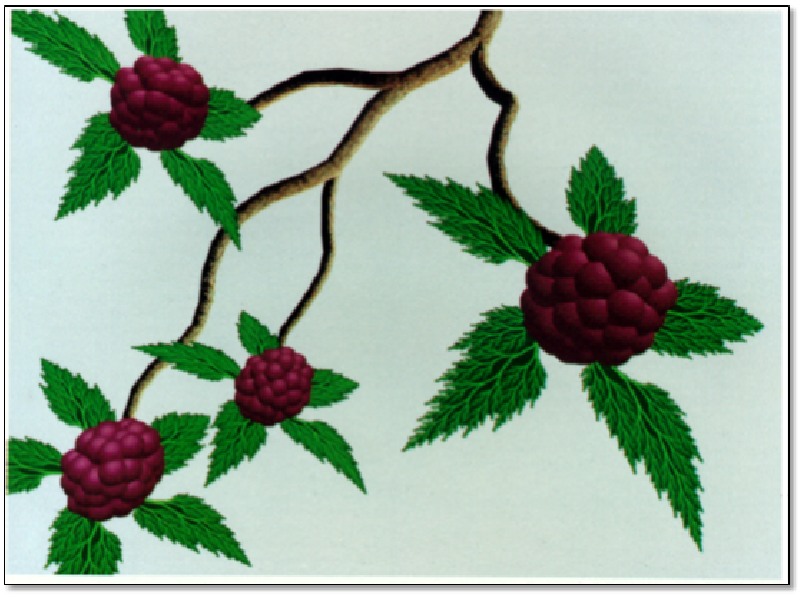


Figure 2.8: Oppenheimer's generated tree branch using fractals [WP95].

Adopting the model of Oppenheimer's work, a tree modeler was conceived and presented by [WP95], where the main idea consisted in using a primary branch which splits a number of times along its length. Each branch may have additional branches similar to the primary one but each individual branch can adopt different attributes from its parent. This differentiation of attributes made it possible to construct a wider variety of plant models.

L-Systems (an acronym for Lindenmayer systems), were developed by Aristid Lindenmayer for the simulation of plant growth and organism development [PL96]. L-systems are parallel rewriting systems (variants of formal grammars) based in the concept of string rewriting. By using a set of production rules, they are able to create complex objects through the successive replacement of more simple ones.

L-systems can be described by three main components: V, the alphabet, a set of symbols containing replaceable elements (also known as "variables"); ω, the axiom, a symbol string that define the initial state of the system; and P, a set of production rules, defining how variables (the predecessor part of the rules) can be replaced with combinations of other variables (the successor part of the rule). The process starts with the provided axiom ω, which is rewritten using multiple rewriting rules P. The process continues with the iterative application of the rules over the resulting strings at each step. The basic graphical interpretation of L-systems is achieved using the so-called Turtle Graphics interpretation, where each symbol is associated to a command in this language. Figure 2.9 shows an example explaining the procedure.

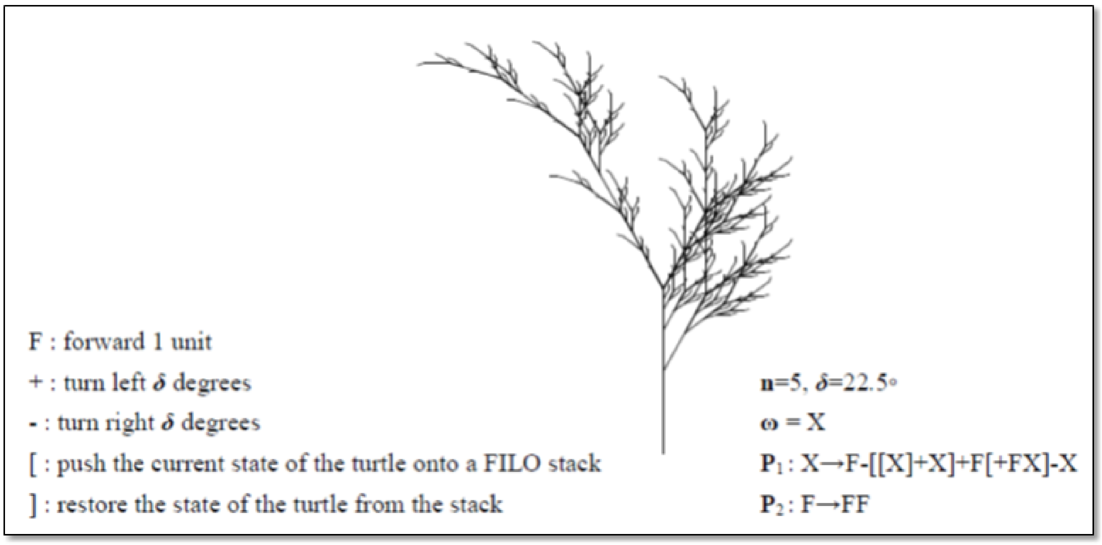

F : forward 1 unit
+ : turn left δ degrees
- : turn right δ degrees
[: push the current state of the turtle onto a FILO stack
] : restore the state of the turtle from the stack
n=5, δ=22.5°
ω = X
P_1 : X→F-[[X]+X]+F[+FX]-X
P_2 : F→FF

Figure 2.9: Tree generation and visualization using Turtle graphics [KM06; PL96].

As pointed by [LD99], Lindenmayer's approach, while intuitive for biologists, proves to be difficult to achieve a specific result by changing the global aspects of the shape. In their work, they aim at combining the power of a rule-based methodology with a more intuitive interface. This is achieved by using a graph description, where its nodes are components that represent parts of a plant and the edges correspond to creation dependencies. Through a graph transversal, these components are generated and placed in an intermediate graph that is used for geometry generation. This approach has been used in the author's conceived commercial modeling software, XFrog (see Figure 2.10).

Among other works, this software has been employed in [Deu+98], which aimed at simulating an ecosystem model to distribute vegetation elements, through the use of height and water maps, as well as the ecological properties of the plant species (e.g. growth) and rules for soil, sunlight and water competition rules. With similar goals, [Ham01] presented an ecosystem-based

Figure 2.10: Vegetation modeling with Xfrog [**Xfro11**] (left) and Speedtree [IDV11] (right).

procedure which relied on elevation data, relative elevation, slope properties and fractal noise to select an ecosystem. This decision determines the vegetation textures and plant cardinality, but their location is randomly chosen. A common problem in dealing with detailed vegetation objects is rendering, which has lead researchers to consider various types of geometrical plant models with different types of complexity. A survey concerning these different types of representations and the ways to use them has been presented in [MG06; SD05]. Detailed vegetation has, in the meanwhile, been intensively employed in real-time applications, such as games for quite a while, using software such as Speedtree [IDV11] (see Figure 2.10). When the reproduction of real-world urban environments is the goal, the employ of existing real-world data is always preferred. Location and type of vegetation can be often found in GIS, but more details are seldom included. It is therefore important to employ some of the referred methodologies, both in generation as in their automatic placement.

2.3.4 Road Networks

Virtual environments that sit on top of the surfaces terrains mentioned in section 2.3.1. are therefore inherently subjected to the difficulties of object placement on top of such, often irregular, terrains. Such problems arise especially when streets and building are to be added, which often has implications regarding their layout. For instance, in the city of Porto, Portugal, whose location next to a river and the sea has affected the terrain elevation, it may be noticed how the surface irregularities contributed to the street network layout, building distribution and building architecture. Such aspects still constitute a challenge for its virtual procedural recreation.

Bruneton and Neyret [BN08] presented a method that enables the detailed integration of various elements such as roads, rivers and lakes in large sized areas, combined for such a digital elevation model of the vector field with information of various objects in order to achieve a more realistic junction (see Figure 2.11). In order to operate on extensive grounds with high level of detail, a scheme based on a view dependent quadtree approach is used, where detail is refined when on the camera distance diminishes. New quads are generated when needed and cached on the GPU. Depending on the required level of detail, the vector information is rasterized in the appropriate resolution, thus allowing the correct adjustment of the various elements of the surface.

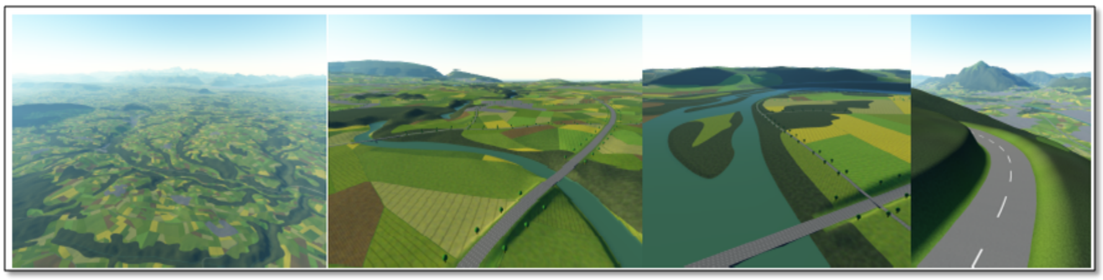


Figure 2.11: Placing rivers, roads and bridges in large and irregular surfaces [BN08].

However, this approach, although very suitable for the placement of roads (a key element in urban environments) still does not address buildings, whose large dimension also presents major challenges in terrain adaptation.

In urban area conception, streets are mainly built first, defining the building positioning in a certain area. For that reason, in some approaches, modeling these elements has been done in a first step [PM01; SMS06].

Parish and Müller [PM01] presented, in a first version of their CityEngine application [Esr13], a system capable of modeling large urban settings through a relatively small set of statistical and geographical data, but enabling the definition of further sets of user-defined rules. This information is then used to define the parameters of L-systems, which in turn proceed in the creation of the road networks. L-systems, originally employed to simulate organism growth (see section 2.3.2.) were employed in street modeling, due to the structural similarities they possess [PM01].

Using the conveniences provided by parallel production nature of L-systems, Parish and Müller [PM01] were easily able to produce schemes of branching roads, once the production rules had been created (see Figure 2.12). Starting from a small segment of road, new segments are procedurally added, reproducing therefore a network of roads in a similar fashion to plant growth. However, despite the ability of this algorithm to produce high quality results, the possibility to make local changes and introduce feature areas is limited to parameter tuning, which is not easily achieved.

In this sense, an alternative was presented in [Che+08], in which the user defines a tensor field that guides the generation of the street network. An example allowing a better visualization of the process is shown in Figure 2.13.

Also aiming at a more precise control was Kelly et al. solution [KM07] in his CityGen

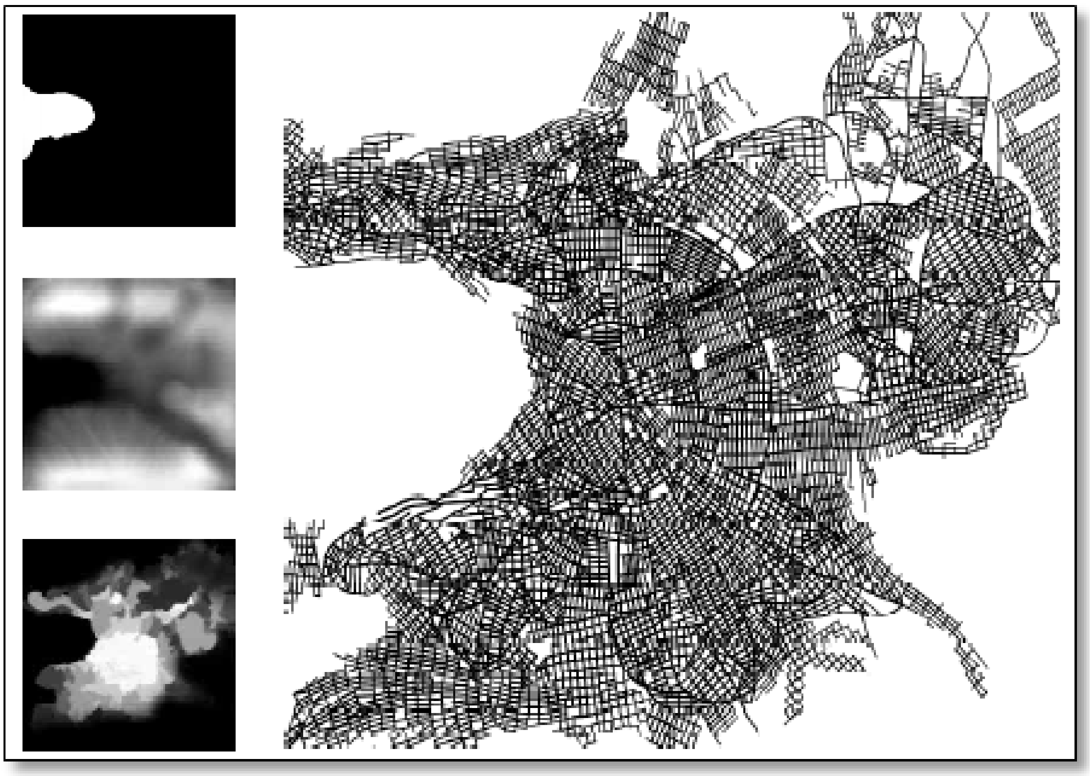


Figure 2.12: On the Left: Maps with water, elevation and demographic density. Right, example of a street map generated out of these data sources [PM01].

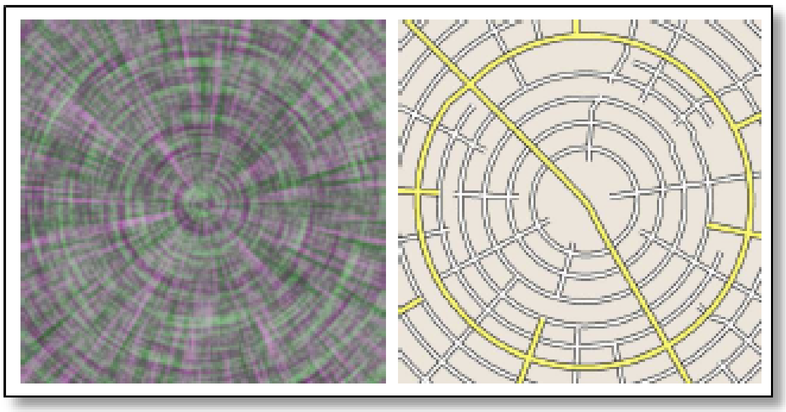


Figure 2.13: Tensor field (left) used to procedurally generate a street pattern (right) [Che+08].

modeling system. This approach seeks to be more interactive in the way that it allows the user to make changes to various parameters, while watching the results in real time. The working scheme consists in the initial generation of the main roads and motorways, being the secondary roads then generated from the former.

The main road network is represented by an undirected graph and by linked lists, using nodes that can be set and changed by the user in real time using the generation application. The roads are then generated between the nodes, adapting themselves to the terrain on which they lie. The generation of secondary roads starts by dividing the space into cells, then filling these areas with streets through L-Systems. The procedure can be modified to generate several different types of streets (see Figure 2.14).

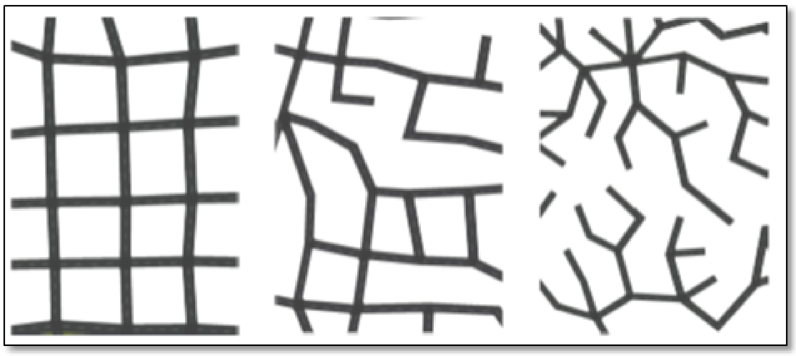


Figure 2.14: Design patterns of existing CityGen streets: Raster, Industrial and Organic [KM07].

Still seeing major limitations of the previously mentioned editing approaches to control the trajectories in to obtain realistic roads, Galin et al. [Gal+10] presented an algorithm for generating roads connecting an initial and a final point adapting to the characteristics of an input scene. This is achieved by computing the shortest path, connecting those points, minimizing a cost function that takes into account the slope of the terrain, rivers, lakes, forests, etc. This path is first discretely defined, and only afterwards converted into a set of piecewise clothoid splines representing the trajectory of the road, which, in turn, is segmented according to the terrain elevation and existing rivers. These segments are then divided into surface roads, tunnels and bridges (see Figure 2.15). The control is thus accomplished through parameterized and controllable cost functions that operate based on the scene parameters.

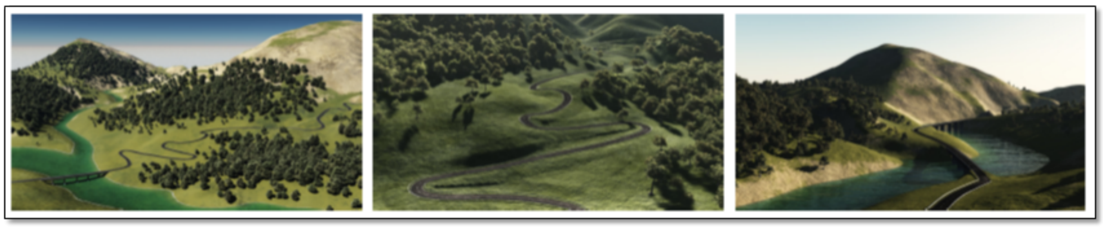


Figure 2.15: A mountain road following the path of a river [Gal+10].

2.3.5 Buildings

In the various works mentioned in the previous section, the generation of road networks is the first step modeling of urban environments, being followed by building generation on the lots formed by the road intersections.

The approach in [PM01] presented also a solution for building generation through L-Systems, in which they consist of simple primitives, recurring to shaders to achieve more detail in the facades. However, according to Wonka and Müller [M+06; Won+03], L-systems are by nature not a good alternative for such an end, since they are more indicated for open-space growth.

Buildings, on the other hand, have more strict space restrictions and do not reflect a growing process.

In [Won+03], Wonka introduced the split grammars, a type of formal grammar which is based on geometric shapes - based on the concept of shape brought up by Stiny and Gips [Sti80; SG72] - for modeling of architectural structures. Starting from a simple non-terminal shape and a set of production rules, shapes are iteratively split and replaced by in multiple smaller shapes, stopping when terminal shapes, such as windows or other wall ornaments, are reached (see Figure 2.16).

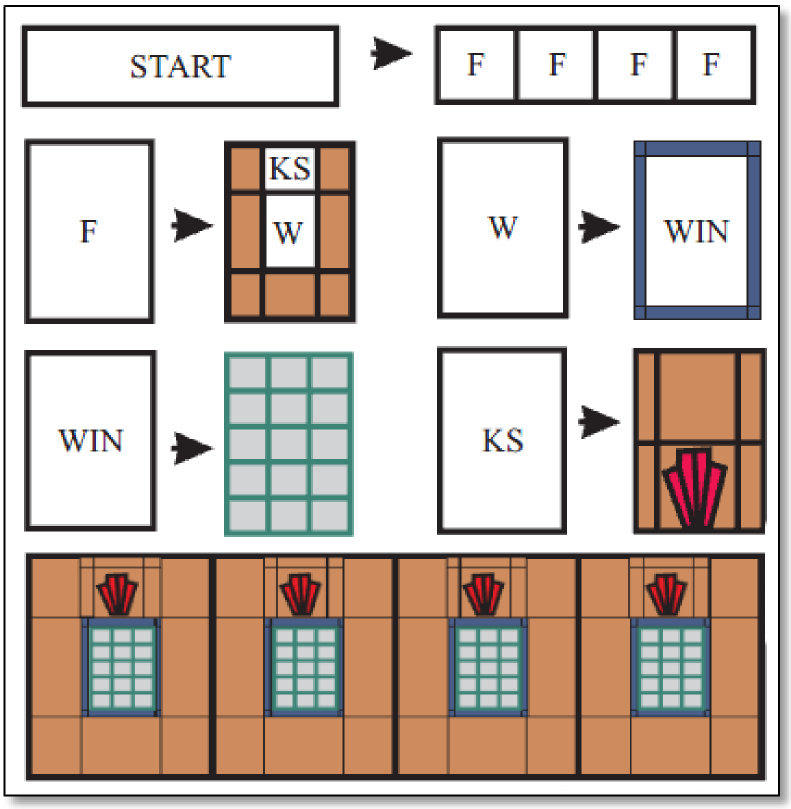


Figure 2.16: Working scheme of split grammars, being the START shape the starting point and the set of windows on the bottom the final result [Won+03].

Using a database filled with large amounts of rules, a variety of results can be produced, without the need to define a grammar for each individual object to be modeled. Due to the complexity of such a grammar, there are multiple rules which can be chosen in each step of derivation. Therefore a second grammar, called control grammar, is used to enable consistency concerning the choices made at each step, forcing all elements of the houses follow a common style.

The approach suggested by Wonka allowed therefore the procedural modeling of buildings with a high level of geometric detail. However, the use of split grammars only apply to simple mass models. In order to overcome this limitation, Müller introduced the concept of CGA shape, which procedurally generates mass variations of the model using volumetric shapes (cubes, cylinders, etc...) at the beginning, proceeding to the generation of detail consistent with the model [M+06].

The grammar works with the configuration of shapes. A shape consists in a symbol (a string), geometry (geometric attributes) and numerical attributes. Shapes are identified by their symbols that can be terminals or nonterminals. The production system proceeds through an iterative evolutionary modification and replacement of shapes. Each rule is given a priority, and the highest priority rule is selected, which guarantees that the derivation proceeds from low detail to high detail in a controlled manner. As in Chomsky grammars [Cho56], the application of production rules is done sequentially.

The mass models can be viewed as the union of more simple geometric shapes. Thus, by applying a set of geometric transformations on primitive blocks, more complex geometric shapes can be created.

There are two mechanisms which aim on generating more consistent quality models: testing the spatial overlap and testing nearby important lines and planes in the shape configuration. The spatial overlap (or occlusion) checks if a shape is visible, partially hidden or totally hidden by others. The line/plane test avoids situations such as the overlap of a window by a wall, i.e. architectural and aesthetical situations that do not occur in real environments.

The application of this new creation process of virtual urban environments can be found in the latest version of the CityEngine [Esr13] system mentioned before. The produced results are amongst the most detailed achieved so far through procedural modeling methods (see Figure 2.17).

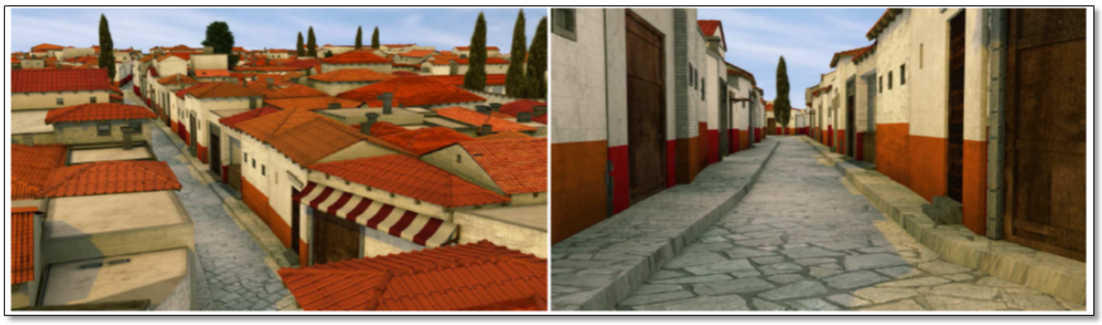


Figure 2.17: CGA Shape application for modeling detailed buildings [M+06].

A generalization of such grammars, named G^2, was presented in [KPK10], introducing the possibility to manipulate multiple types of non-terminal objects, as well as to pass non-terminal symbols as parameters into rules using the definition of abstract structure templates. This allowed the control over a more diverse geometric representations, which proved useful to generate other elements besides buildings, such as trees (see Figure 2.18). Abstract structure templates, on the other hand, provide greater flexibility and prevent rule explosion more efficiently.

One important limitation of all these top-down approaches is that any splitting strategy leads to independent parts, which cannot be joined afterwards. This limits the construction of

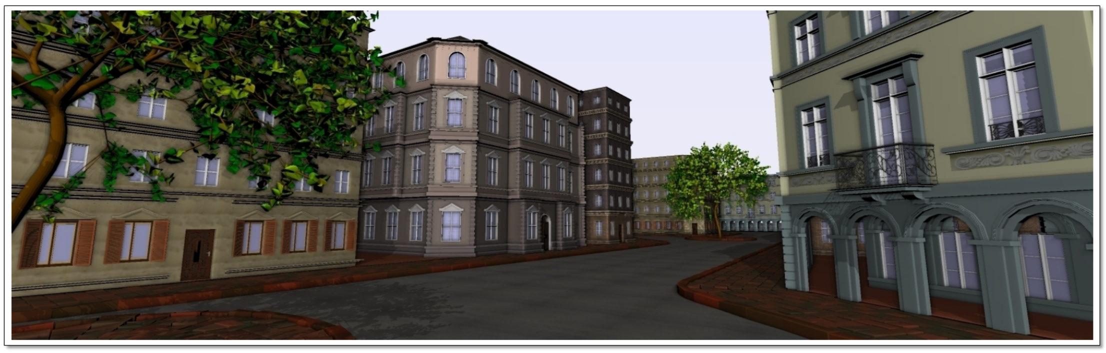


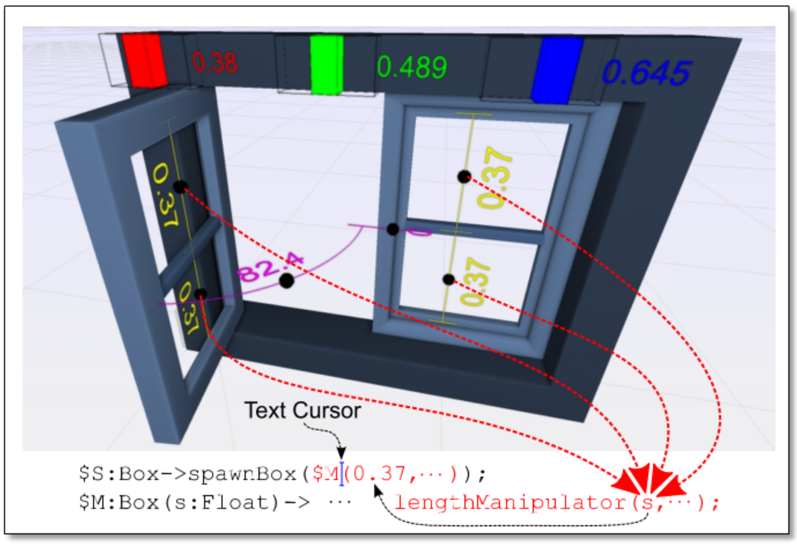
Figure 2.18: Generation of a street featuring houses and trees, achieved by employing different non-terminal symbols to better address each type of object [KPK10].

structures that require connections between objects. In an attempt to address such deficiency, the approach in [KK11] presented a method to generate interconnected structures such as bridges, roller coasters and other building connections, focusing on functional interactions between rigid and deformable parts of objects. The process consists in collecting attachment points that are then analyzed and elected using abstract connection patterns or geometric queries.

In the previous works, shape-grammar based procedural techniques were successfully applied to the creation of architectural models, but since they were text based, they could be impractical to use by most artists. Also, the control over the final model was only possible to achieve by changing the grammar rules or the global parameters. This problem has been addressed in [LWW08], where traditional modeling techniques were introduced in a visual editing system, with direct local control over each grammar aspect.

Considering that one would want to change the appearance of a specific window on a facade, the text based system would require writing multiple rules to identify the floor and column of the facade, in order to apply the changes. On the other hand, using a visual editor, the user can directly select the intended window to apply the changes. This selection can be extended to multiple elements of the same type or that have a common meaning, using semantic attributes such as facade number, floor or column. All changes, produced using this method, are also stored persistently, using strategies to avoid input losses whenever substantial changes to the underlying grammar are performed. Despite these advantages, this interactive approach still required a deep understanding and maintenance of the rule-based structure of the underlying grammar.

Building on this observation, simplified interaction schemes have been made possible through 3D manipulators [KK12], which provide a visual representation of adjustable parameters directly in model space, revealing their influence (see Figure 2.19). To avoid cluttering and to drawn the user's attention to a specific subset of manipulators, modules of the procedural grammar can be associated with a set of camera views. These concepts are encapsulated into procedural high-level primitives to support an efficient procedural control over 3D data within a High-Level mode. However, these primitives must be defined in a so-called Professional-mode, which still requires editing a complex text-based grammar.



```
$S:Box->spawnBox($M(0.37,···));
$M:Box(s:Float)-> ···   lengthManipulator(s,···);
```

Figure 2.19: Parameter manipulation using interactive 3D manipulators [KK12].

A popular alternative for the textual rule definition is the visual programming language paradigm, especially the one based on dataflow graphs. The work of Patow [Pat12] is a general system with the goal of making shape grammars more comprehensive by employing a visualization in the form of a node-based system (see Figure 2.20). Making use of the directed acyclic graph representation already provided by the Houdini engine [Sid15], this approach bridges the disassociation between rules, their interrelationships, and their applications. The work of [BBP13] complemented the previous approach by providing an end-to-end system for procedural copy & paste, introducing a graph-rewriting method for seamlessly gluing source and target graphs and obtaining consistent new procedural building rulesets.

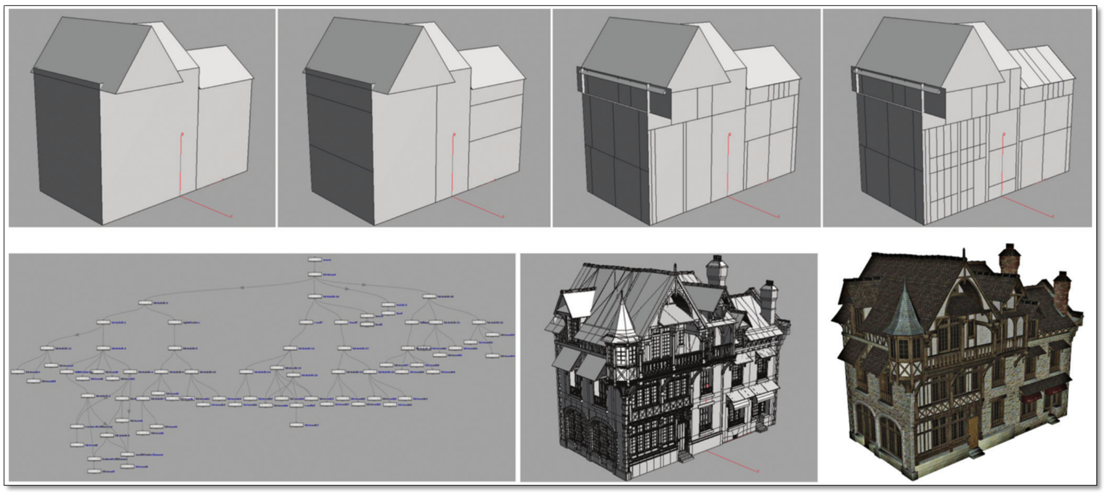


Figure 2.20: Different construction stages of a house generated using a graph-based approach [Pat12].

Another interactive, non-grammar related, procedural modeling approach has been presented by Kelly [KW11], which allowed the reproduction of difficult architectural surfaces (e.g. curved or overhanging roofs, buttresses, chimneys, columns, etc.) through procedural extrusions. By defining a building footprint outline and a set of vertical profiles lines, a sweep algorithm can be applied, merging the various features into a realistic building structure. Specific features can be added through the definition of anchors and plan edits. Modeling larger environments has been achieved by example, and by using floor plans from GIS databases (see Figure 2.21).

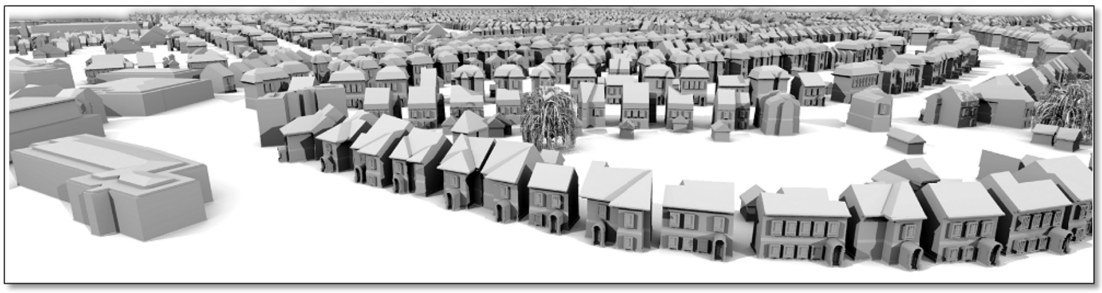


Figure 2.21: Procedural extrusion approach on building floorplans from GIS [KW11].

Another, unrelated, yet very interesting approach was presented in [Mer07] and is based on the concept of model synthesis, similar to the texture synthesis technique. The idea consists in partitioning a small example using a three-dimensional grid. This partition then generates automatically a set of constraints: two parts can only be adjacent in the generated model if they are adjacent in the example model. This approach can be used to create symmetric models, models that change over time, and models that fit soft constraints. Although this is quite limited due to its grid layout (which in the creation of urban areas is rarely applicable), the results are quite complex but simple to create.

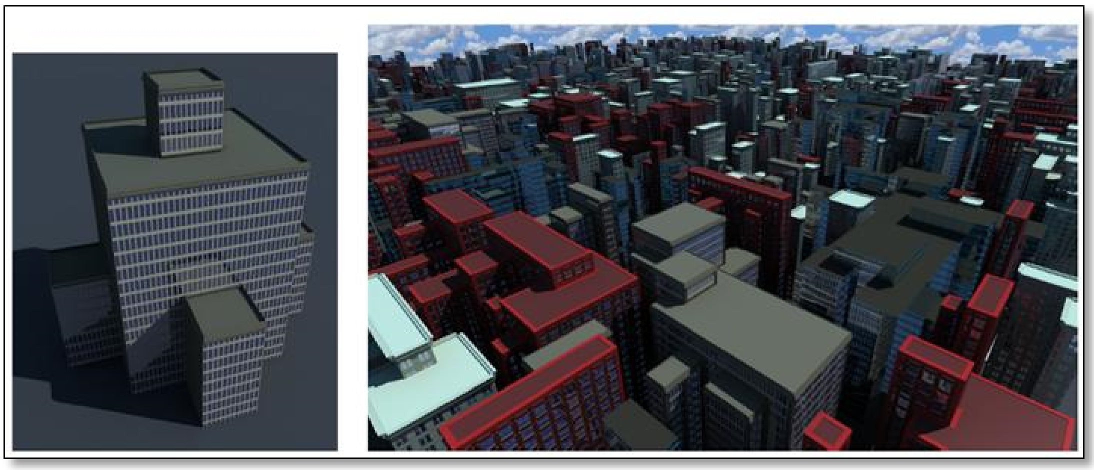


Figure 2.22: Use of an example model (left) to generate complex cities (right)[MM08].

This approach has been improved in [MM08; MM11] (see Figure 2.22), introducing several geometric constraints: dimensional, which allows objects to have predetermined dimensions; algebraic, which introduce algebraic relationships between their dimensions (such as having the height twice the length); incident, which allows the management of incident vertices to faces; connectivity, which indicates how objects should be connected; and large-scale constraints, which allow the introduction of general ideas of how the model should look (similar to the soft constraint model synthesis presented in [Mer07]). This new approach brought up several major improvements to model synthesis that allow the user to control the output more effectively.

Also aiming at a more intuitive control over the procedural processes, Smelik [Sme+11] presented a declarative approach which allows designers to operate based on higher-level descriptions instead of complex model generation rules. This is achieved by using the potential of semantically rich objects, as described before. Through the introduced concept of procedural sketching, users interactively create a 2D digital sketch, i.e. a rough layout map. Each sketched element is procedurally expanded to a corresponding terrain feature (see Figure 2.23). Through a semantic model for virtual worlds, the terrain features are classified according to their nature and can be used to automatically generate further details. The obtained models are enriched with relevant semantic information on their functionalities, services and roles. Another contribution of this work lies on the consistency maintenance, which guarantees semantics coherence of the modeled terrain features. For instance, each feature fits itself to local constraints, affecting nearby structures or connecting to any compatible feature, so as to avoid unrealistic features. The contributions of his work are implemented in a prototype called SketchaWorld.

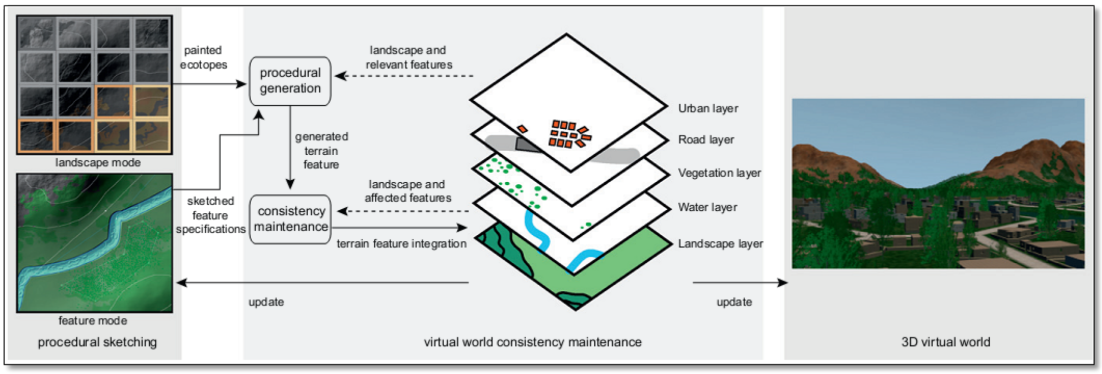


Figure 2.23: Generation and consistency maintenance of procedurally sketched terrain features [Sme+11].

Despite the indicated advantages proposed by Smelik's work, some limitations and unsolved challenges remain due to the high-level nature of such declarative approach. When more fine-grained manipulation is intended (such as the edition of geometric meshes or manual placement of certain features in less plausible places), some difficulties arise, derived from the common problems of integrating manual and procedural approaches. These are thoroughly described in [Sme+10].

Another approach for interactive control of 3D urban models was proposed in [VAB12] through the concept of inverse procedural modeling. Here, users can specify arbitrary target indicators, while the system is left to estimate the procedural parameters and rules needed to generate the intended output. Presented indicator examples concerned sunlight exposure per facade, distance to nearby parks and floor-to-area ratio of several buildings (see Figure 2.24). To explore the parameter space and search for valid solutions, a strategy using Monte Carlo Markov Chains and resilient back propagation was employed, together with algorithms to search both local and global state changes. Coupled with a forward modeling process, the goal was to provide both increased control and efficiency in generating models that satisfy user requirements.

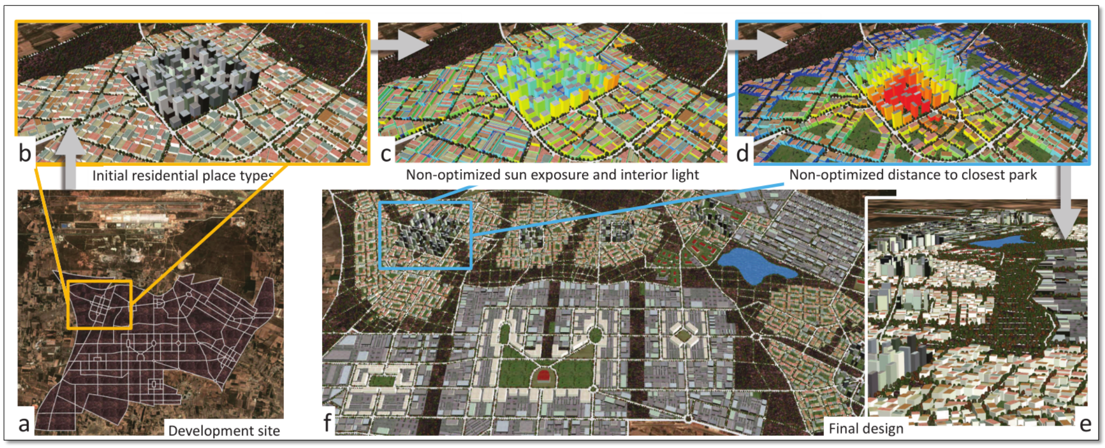

b
Initial residential place types
c
Non-optimized sun exposure and interior light
d
Non-optimized distance to closest park
a
Development site
f
Final design
e

Figure 2.24: Interactive control of a 3D urban model, possible by altering the parameters of an underlying procedural model (forward modeling) or by changing the values of arbitrary target indicator functions (inverse modeling) [VAB12].

An older, yet very important contribution derives from the work of Greuter et al. [**GPSL03b**; Gre+03], which presents a framework for generation of entire virtual worlds, to which he calls 'pseudo infinite' due to the fact that the presented virtual cities have no apparent boundaries and can be explored to a pseudo infinite extent. This is achieved by constantly generating new buildings in real-time, based on the camera position and its view-frustrum, which restrains the number of buildings. In subsequent frames, buildings are cached to avoid regenerating them on each draw call. When the buildings drop out of viewing range, they are deleted in order to reduce memory consumption. In order to guarantee that the exact same building appearance is produces when it is looked at again, generation is guided by a global seed value and optional parameters, which assures that generation occurs always the same way for a building located in a certain position. The presented results correspond to an extremely large city, consisting of 4x1018 geometrically different buildings [**GPSL03b**; Gre+03]. The process that takes place is depicted on Figure 2.25.

The previously mentioned approaches have proven to be suitable for the procedural modeling of fictitious, yet geo-typical, i.e. realistic urban environments, as they present impressive levels of detail, as well as the possibility of creating a large variety of buildings. However, when geo-specific, i.e. real-world cities are meant to be reproduced, there is the need to use some kind of real-world data.

In order to overcome this problem, the integration with geographic information systems has been suggested by some authors [Coe+07; DB05; SMS06], since a lot of georeferenced information of urban area can be automatically extracted. Thus, it is possible to model buildings in real positions and sizes without the need for human intervention. While CityEngine [Esr13] does support loading of multiple types of geographic data, it does not provide, in some cases, advanced native support to deal with such data, meaning that much semantic data relations are hard to extract. In the work of [Coe+07], the concept of geospatial awareness is presented in

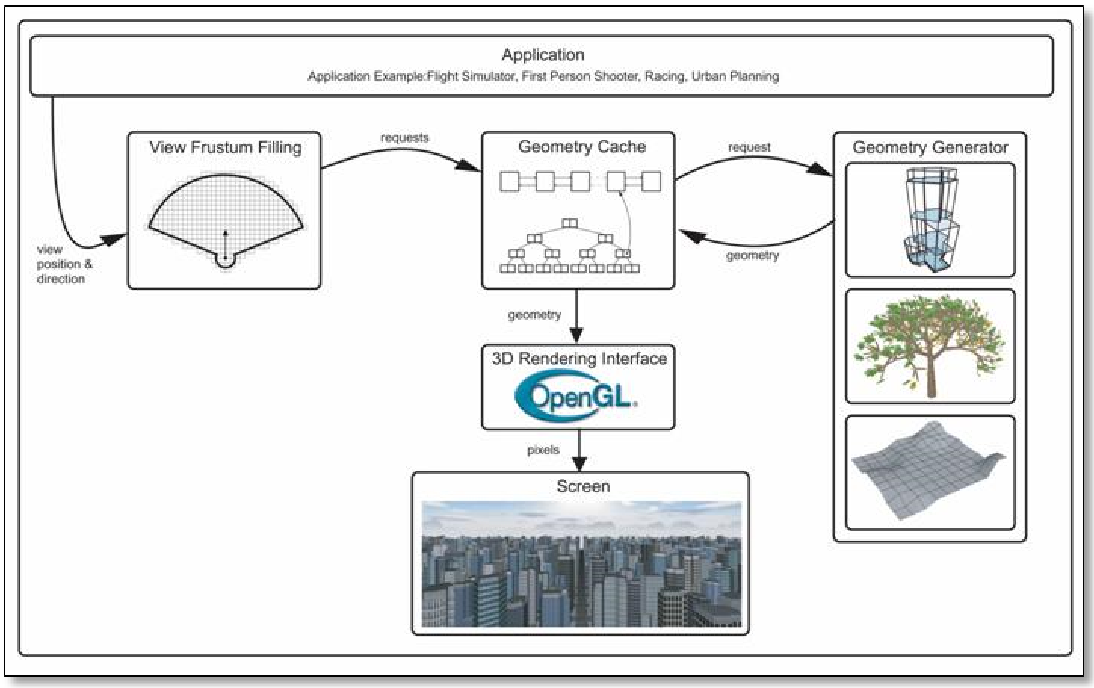


Figure 2.25: Process of generation of 'Pseudo Infinite' Cities [**GPSL03b**; Gre+03].

scope of urban generation. This concept refers to the ability to perceive spatial relationships between the various urban elements and their surroundings, thus allowing the definition of some basic rules of construction. For example, buildings whose walls are too close together should not possess balconies. This idea was incorporated in Geospatial L-Systems, which are an extension to parametric L-systems. Thus, the data amplification nature of L-systems is joined with the geospatial systems, which allow the spatial analysis and indexing of georeferenced data. This solution is embodied in a tool - the XL3D modeler - which integrates various sources of information and hence allows the reproduction of existing urban environments (see Figure 2.26).

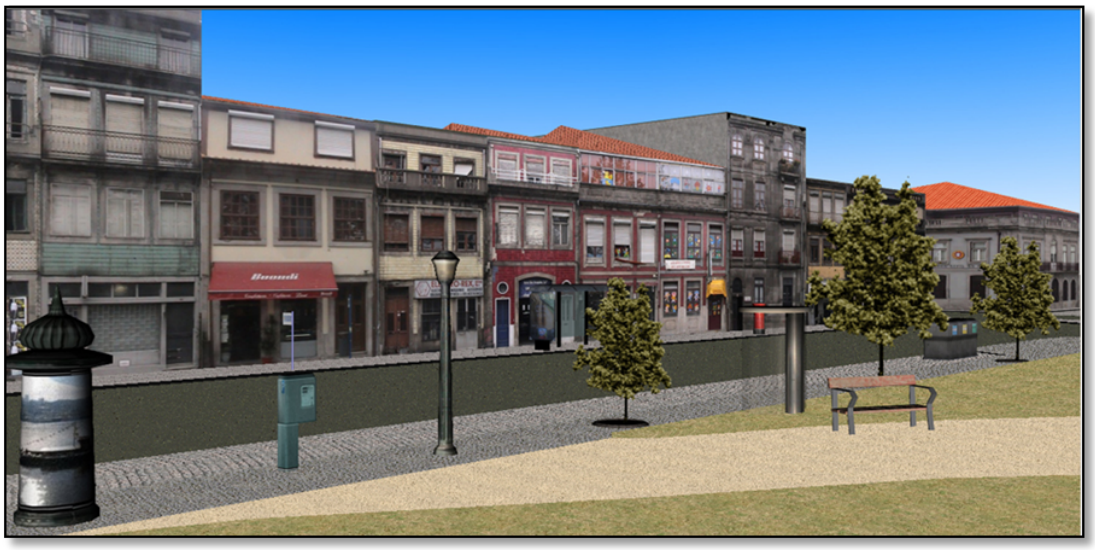


Figure 2.26: View over a reproduction of the Praça da República, in Porto, Portugal [Coe+07].

Based on Coelho's work, Silva [SC10] developed the PG3D modeler, a solution for procedural modeling of virtual urban environments for digital games, whose main characteristic lies on its implementation directly in spatial database management systems, which serve as the storage location of the information sources regarding existing urban areas. The modeling processes, working through a set of internal stored procedures in the database, are able to develop virtual environments with real content and also save their generated models in the database. Thus, the entire data access is done internally, reducing the time required for their accomplishment.

The modeler operates based on PG3D grammar, which is an extension of shape grammars [Sti80; SG72] and, more specifically, the CGA Shape [M+06], endowed with the capabilities of geospatial awareness and relational development, stemmed from Geospatial L-Systems [Coe+07]. Having been strongly influenced by both, it constitutes an attempt, to combine their potential. It is used for the definition of the production rules and structures by which the modeling processes operate for the creation of urban environments. These structures are called PG3DShapes (inspired by the CGA Shape) and can illustrate a surface, street, building, or even just a portion of it, such as a wall or corner of a window. A shape contains a set of geometries that graphically describe the type of element to be represented. These data structures, together with database querying abilities make it easy to create optimized polygonal meshes, allowing large scale urban environments to be used in computer games (see Figure 2.27).

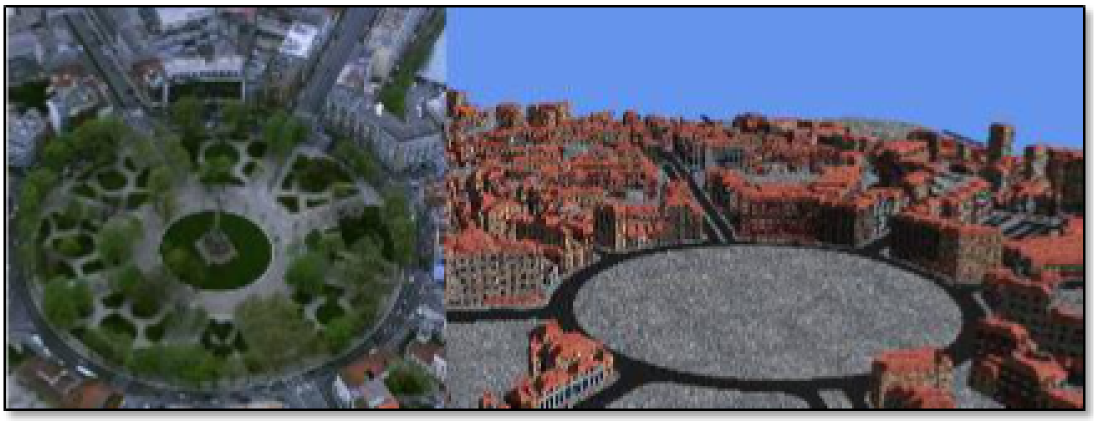


Figure 2.27: Comparison of the Boavista Roundabout in Porto, Portugal and its correspondent virtual environment [SC10].

2.3.6 Building Facades

There are several characteristics that, in an urban environment, make each building unique and distinct from their counterparts, such as their volumetric shapes, spatial orientation, as well as the arrangement of the various elements in their facades, whose details are sometimes essential to make a difference.

In the virtual recreation of an existing urban environment, it is essential to introduce these characteristics in order to give the viewer a feeling of familiarity. In this sense, some authors have presented more detailed procedural modeling solutions, focusing on the realistic representation of the facades.

As a further development to his previous work, Müller presented a system that, given a low resolution image of the facade of a building, is able to create a high resolution three-dimensional model, possessing strong similarities with the introduced image [M+07]. Additionally, a semantic interpretation of the images is performed, allowing the inference of grammar rules (see Figure 2.28).

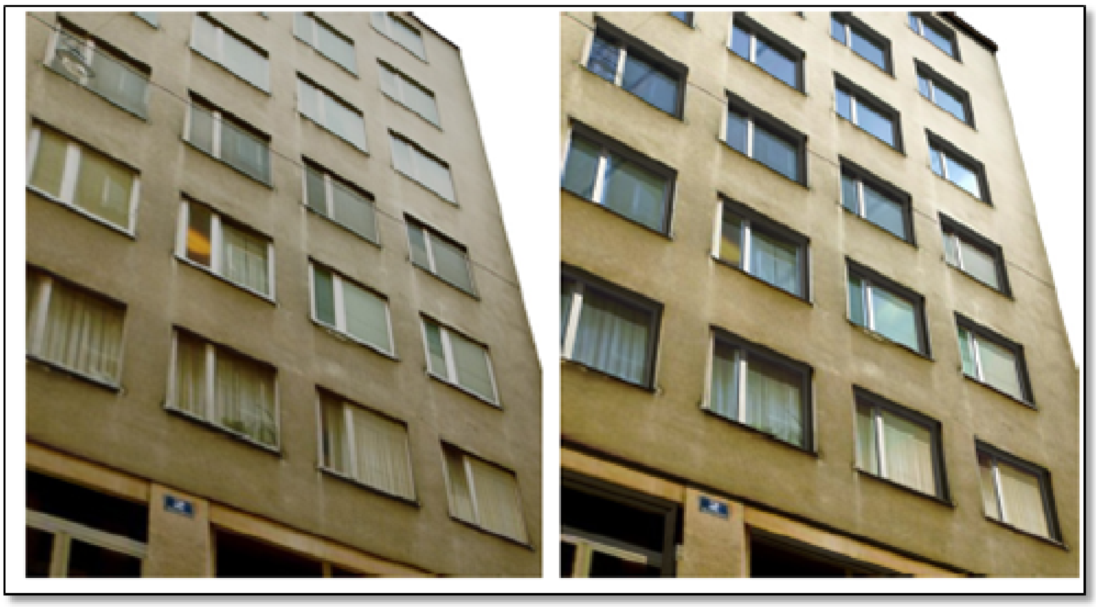


Figure 2.28: Left: Photograph of a building facade. Right: three-dimensional model of the facade [**MZWG07**].

The process is divided into four distinct steps [M+07]:

1. Detection of facade structure: Subdivides the facade image in floors (vertically) and tiles (the divisions within a horizontal floor) through the detection of mutual information. Tiles are important concepts in procedural modeling, symbolizing architectural elements such as a windows or doors, together with the surrounding walls.

2. Section refinement: Once the facade division into tiles has been achieved, where the most similar are grouped, tiles are segmented into smaller rectangles. This step derives from split grammars.

3. Element recognition: The small rectangles are compared to the three-dimensional objects of an architectonic element library. Afterwards, the generation of three-dimensional textured model follows, together with its semantic structure in a shape tree.

4. Edition and Extraction of grammar rules: The semantic interpretation of facades may be used in various edition operations, including the extraction of grammar rules from the created shape tree.

Despite the high quality results and the great level of visual fidelity between the images and generated models, the image processing system is very sensitive to any noise existing in the images, hampering its use in many cases. Another procedural modeling approach for facades was presented by Finkenzeller [Fin08], who aimed at producing much more detailed facade elements than Müller, such as frames, borders and ornaments which often can be found in buildings (Figure 2.29, b).

This approach requires much greater activity coming from the user, who is responsible for sketching a coarse model of the building. However, this specification is intended to be fairly simple, serving merely as a way to establish the contours of the building, as well as its purpose and intended styles (Figure 2.29, a).

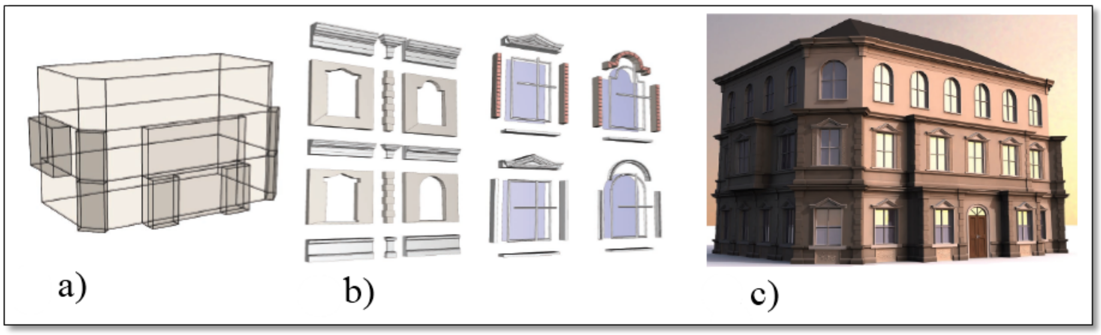


Figure 2.29: Facade modeling process suggested by Finkenzeller [Fin08].

Afterwards, the system extracts all the spatial information of all facade elements and automatically identifies and adapts surrounding structures, avoiding the need for occlusion calculations, as in [M+06]. Only then are the detailed elements, such as doors, windows, eaves and other ornaments procedurally generated and placed on the facade (Figure 2.29, b)

Lastly, the system saves the facade in a hierarchical structure that reflects its symbolic representation, i.e. generates a parameterization of the facade. As a result, users can change parameters of the facade at a higher level and produce more complex structures in less time (Figure 2.29, c).

2.3.7 Other Urban Content

An urban environment is much more than buildings surrounded by roads. This is why it is important to model other content in these common spaces, such as street lights, benches, traffic lights, signs and other kinds of street furniture [Coe+07; SMS06].

The modeling process of this type of content is relatively simple, in that the variations that occur within each class of objects are simple to categorize. For example, trash bins have only limited amounts of variations, meaning that their models can just be instanced and scattered around the cities. This procedure also applies to most street furniture, meaning that their models or generation instructions may be stored in a database and reused as desired.

Although modeling such content is simple, the same does not apply to their placement and orientation on the terrain, since they follow very specific rules in real urban environments. For example, it does not make sense to place such objects in the middle of the road or in front of house entrances. Also, the orientation of traffic lights or signs should be consistent with the road they sit next to. It is therefore essential that their arrangement is not purely random, but that is driven by a set of rules. Examples of such rules include, for instance, the placement objects within the pedestrian areas and the orientation of bus stations towards the streets.

Another possibility, besides using rule definitions, is the use of georeferenced information that regards street furniture, which may be present in databases and geographic information systems. In his work, Coelho uses both possibilities [Coe+07]. In his approach, geospatial information is used for the correct positioning of the various elements, as well as for establishing relationships between them (geospatial awareness), which allows their proper orientation in space (see Figure 2.30). In case the necessary information is unavailable or insufficient, the use of geospatial L-systems induces data amplification features, by using the mentioned rules as well as a certain degree of randomness.

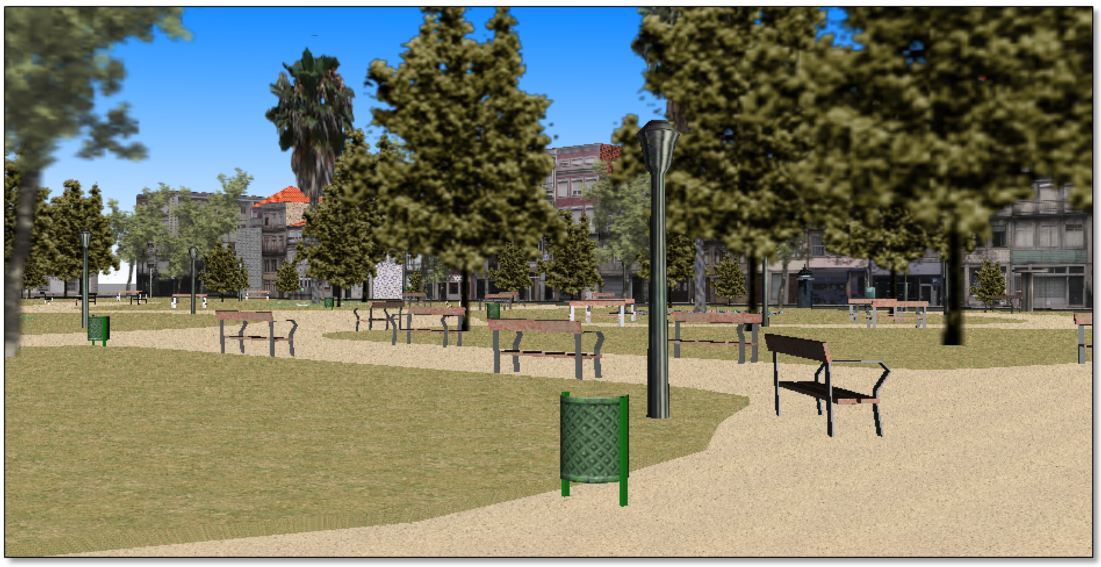
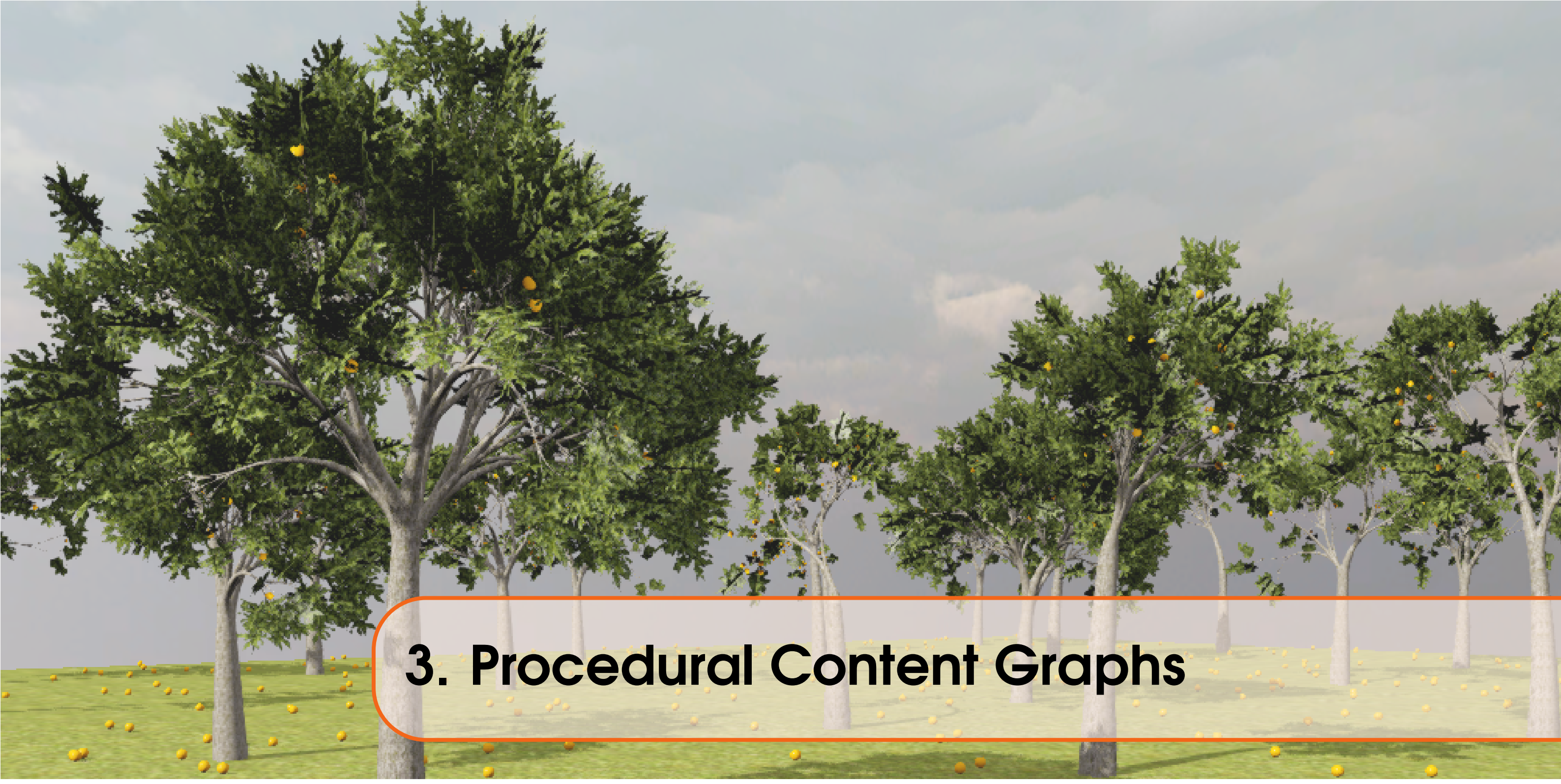


Figure 2.30: View over a green area of urban furniture, presented in Coelho's work [Coe+07]

2.4 Summary

The resort to procedural content generation has become increasingly popular in the production of virtual urban environments, since these are bound to many guidelines and rules that can be easily encoded into algorithms. Grammar and script-based approaches are amongst the most common solutions to assemble and control the rules, yet they have clear limitations regarding readability, manageability, and expressive power when addressing a variety of complex structural designs.

Moreover, such approaches aim typically at geometry specification, and do not facilitate integration with other types of content, such as textures or behavioral entities, which could rather accompany the generation process. In this sense, it is important to analyze how procedural generation control can be improved so as to include the production of application-specific entities and help addressing the enumerated level design challenges.



3. Procedural Content Graphs

The basis of this work is a ***P**rocedural **C**ontent **GR**aph* (PCGR), where nodes and edges describe the procedures and the data flow, respectively. This chapter will first give an overview of the graph architecture, the nature of the handled entity data and the execution process, tackling issues such as recursion, impulse management and synchronization. Then, the control over aspect data will be addressed, explaining the potential behind the control over attributes and how parameters can be defined using expressions.

On a later section, the process and benefits of graph encapsulation will be explained, followed by the description of augmentations, which operate as extensions to nodes, making their definition more flexible and powerful. Afterwards, the possibility for reference management, state persistence and use reflection nodes will be presented as a means to introduce an imperative paradigm into the data flow approach. The concept of dynamic loading will come next as a means to load nodes during graph execution and execute them under specific conditions, attaining more flexible data management features.

This chapter will finalize by explaining how semantics is present on the representation of both entities and procedures, and how it can contribute to a more powerful and higher-level control over the content.

3.1 Graph Structure

A procedural content graph is represented by a directed, cyclic graph $G = (N,E)$ where N are the nodes and E the edges (see Figure 3.1). Its topology encodes a data flow process for content generation: edges are the carriers of content, while the nodes are their operators. Nodes represent procedures that execute series of program instructions that typically (but not imperatively) perform a transformation, analysis or filter on the incoming data, after which they return it. In the graph domain, the terms "procedure" and "node" will be used as synonyms of the same concept.

Different types of data can flow and coexist within the same graph. Their manipulation possibilities depend on the availability of procedures that handle such data. Nodes have typed input/output ports, to which incoming/outgoing edges can connect as long as the ports are of compatible data types. Edge connections are directed, each only linking one output port to one input port and data flow occurs always in that same direction.

The graph structure is meant to allow the sequencing of procedures, where cycles represent recursive operations. After a procedure is executed, its output data is transmitted via the output ports along the connected edges to the subsequent input ports, where the data is stored in a queue. Procedures are executed in several rounds until their data queues are depleted.

Several node types are distinguished (to be explained in further sections), each using a slightly different figure representation. The rectangle, rounded-corner rectangle and octagon refer to standard, encapsulated (see Section 3.5) and augmentation nodes (see Section 3.6), respectively.

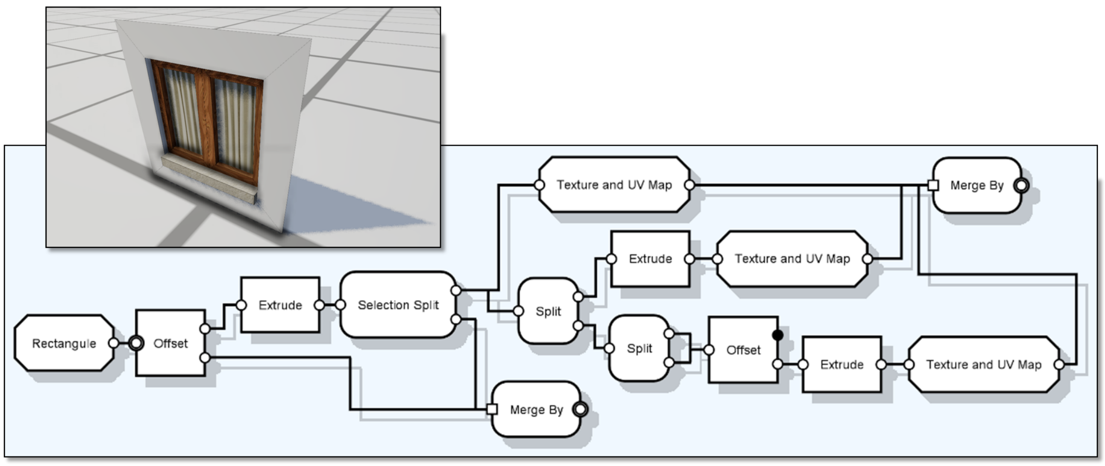


Figure 3.1: A Procedural Content Graph that generates a window.

3.1.1 Ports

There are two types of input ports: single input ports (represented with round outline) will consume one data object per round. Collective input ports (represented with square outline) will consume the whole object queue, handling it as a list. This introduces the possibility to handle sets of objects instead of individual ones. A node that features at least one collective input port is called a *collective* node, as opposed to normal, *single* nodes.

Several edges can connect to one input port (a *convergence* port), where arriving elements are queued for execution. If several edges leave an output port (a *divergence* port), the outgoing data is copied. The result of executing a graph is obtained by collecting all the data emanating from all unconnected output ports, or *leaf* ports. Ports can be blocked (represented with dark fill) in order to discard its results. Blocking is also applicable to non-leaf ports or even edges, which can be useful to temporarily or conditionally prevent the flow at a particular location.

This approach does not pose any restrictions on the number of ports of a node: it can feature zero, one or more input and output ports. The reasoning is the following:

- Multiplicity of input ports allows combination of specific data, according to their type and semantics, making the *conjunction* of flow more explicit and manageable. It is crucial to combine different types and perform context-dependent modifications.
- Multiplicity of output ports allows a natural separation of data according to their type and semantics, making the *separation* and *filter* of flow more explicit and manageable. It is essential for nodes that split entities or simply need to separate the flow based on a certain condition or type of result.

This way, the data flow filtering is encoded in the graph topology, instead of resorting to rule names [M+06], labels [Pat12] or tags [KK12] that can make the development more tiresome, less readable, less manageable and more error-prone, especially for more complex designs.

3.2 Entities

The primary type of information manipulated within a graph, transported through the edges between ports, is organized in the form of *entities* (see Figure 3.2). As their name implies, they represent independent and self-contained objects, carrying their own, specific semantics. Each entity has its own specific data model, which should aggregate the necessary properties in the most adequate and optimal structure.

For instance, to produce and manipulate geometric 3D data, a boundary representation featuring polygonal faces, edges and vertices is adequate for most modeling operations. Yet a set of nodes, enacting crossings, connected by edges, enacting streets, is a far more concrete and appropriate data structure for representing street networks. The flexibility to introduce and manipulate any kind of data type means that more semantically-rich, high-level entities can be employed to produce particular kinds of content. As a result, specific procedures can be developed to optimally deal with such structures. For instance, by using "terrain" and "street" entities, one can apply algorithms, such as the ones described in [KM07] or [Gal+11] that operate over such specific data representations.

Although each entity is expected to have its specific features, all entities share the possibility to incorporate custom *attributes*. Attributes are represented via a hash map in a key-value fashion, and enable us to store properties in an entity dynamically, extending the design of the entity. This process makes it possible to attach properties, such as "name", "index","amount", which may only be relevant and applicable within the scope of a particular graph. When an entity instance

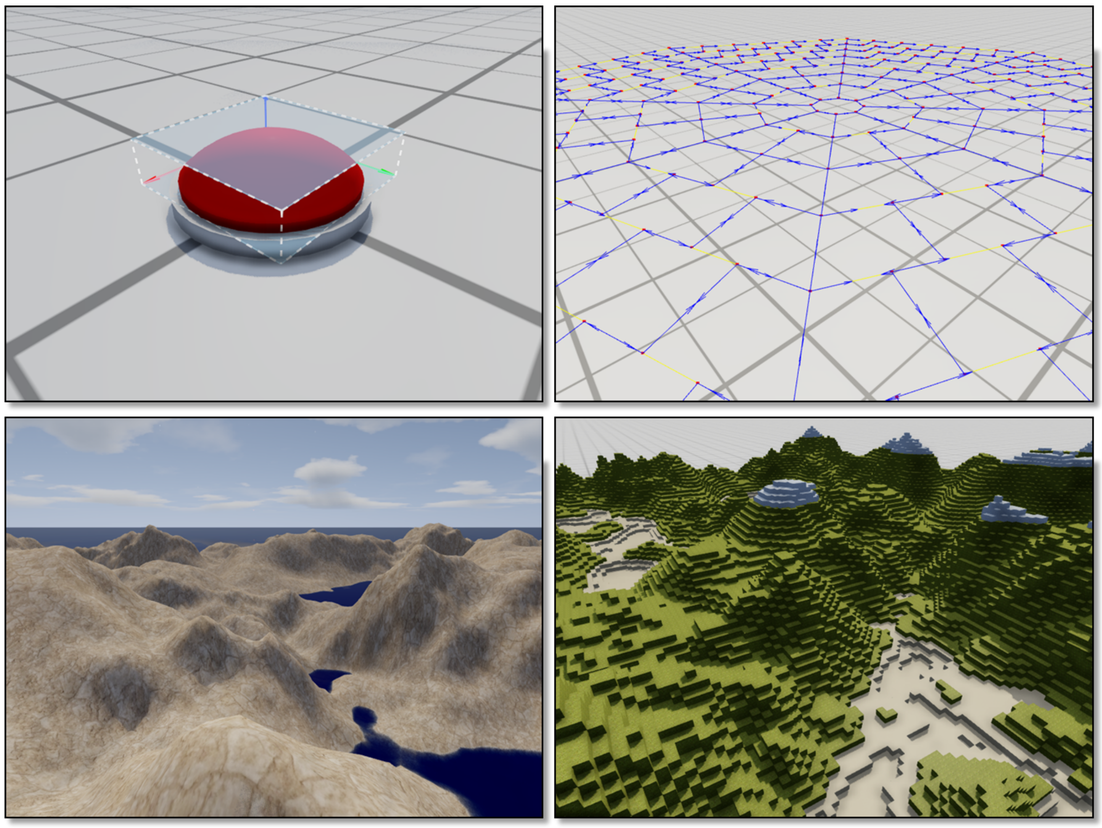


Figure 3.2: Example featuring several types of geometric-based entities in the same image, such as a street, terrain, shape and texture.

is used to create new instances (such as a split that cuts one shape into multiple ones), the new ones are called its descendants, while the original entity is considered an *ancestor*. Attributes are always copied from the ancestors to their descendants. The precise functioning and use will be explained in section 3.4).

Entity types can also derive from others, following an inheritance pattern. This aspect is relevant during the creation of procedures that operate on entity supertypes and not just specific types. For instance, node ports of type 'Entity' can accept any type of entity data. The 'Entity' supertype in itself hold little semantic meaning, but can be used to control the flow of data of other entities and trigger procedure executions (see Section 3.3.2).

Each entity can and will typically aggregate other entities. For instance, the b-rep meshes contains vertices, edges and faces which could be separated to flow individually throughout the graph. Likewise, meshes can also be aggregated within other meshes, recursively. This recursive construction is useful, as it will allow us to group elements (e.g., a city entity can contain a list of region entities, that in turn contain houses, which are represented by mesh data). They can also include details derived from its aggregation: the box scope is a bounding box which contains all the geometry but holds its own orientation.

3.3 Data Flow

The sequence in which nodes should be executed is mainly determined by the topological order of the directed graph, in the sense that the sequence of nodes is based on their dependencies (see Figure 3.3). Hence, the first nodes to be executed are those featuring no input ports (called *source nodes*). For the ordering of the other nodes, two rules apply (the first prevailing over the second)

1. For every edge NM that connects an output of a node N to an input of a node M, N should be executed before M.

2. If a node N has several output ports O_1, O_2...O_n, each connected to a different node M_1, M_2...M_n, then M_{n-1} has precedence over M_n. Among nodes connected to the same output port, no order is stipulated.

In other words, the order is topological and the output port order determines the sequence of execution of the following nodes.

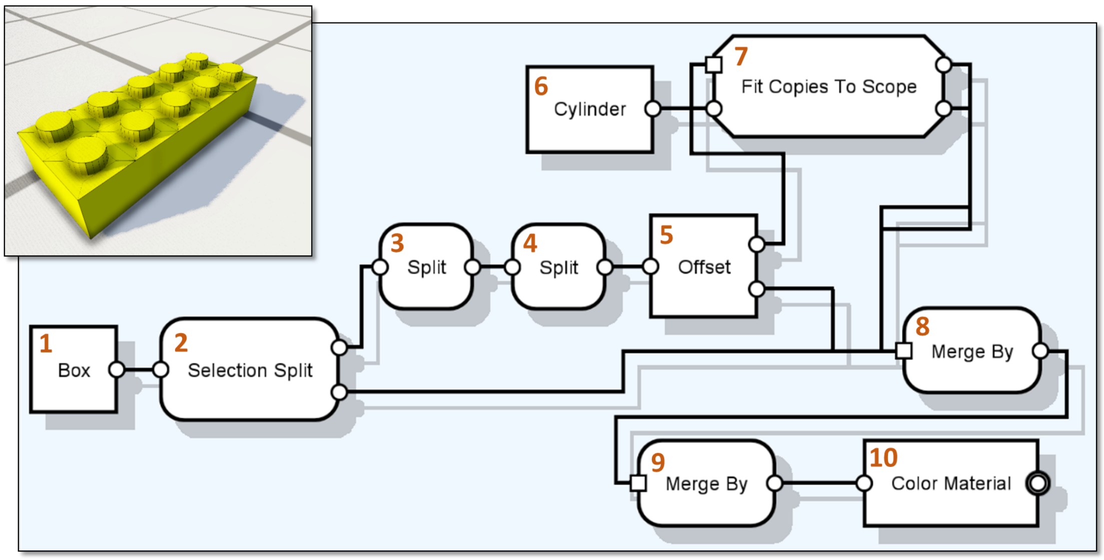


Figure 3.3: Node sequence in the generation of a Lego brick, whereas the node numbering indicates the topological ordering of the nodes.

After the topological ordering of the nodes has been determined, the execution of the graph follows a data-presence protocol. Nodes can only be executed if each of their input ports has at least one entity in every of their input queue. The execution process runs as follows:

1. Add the source nodes to a node queue Q (according to an order which can be user-specified);

2. While Q is not empty:
   (a) Pop a node N from Q;

(b) Execute N according to its nature:
- Source nodes are executed once;
- Single nodes will run several times, each time popping one entity from each of their input ports and executing a round with those entities, until at least one input queue is empty.
- Collective nodes will execute once, since the list of entities will be emptied in one go. This might lead to some entities being left at single input ports.

(c) Pass on entity data from connected output ports of N to the subsequent input ports of nodes M, according to established edge connections;

(d) For all nodes M connected to an output port of N, we verify if all their input port queues are no longer empty. If all are filled, M is ready to be executed and therefore is added to Q, placed according to the topological order.

3. Retrieve all entities from non-connected output ports for storage or display. As mentioned before, entities can also be discarded from the final result by setting the state of their output ports to blocked.

3.3.1 Recursion

Recursion is a ground stone of grammar-based definitions: given its functional paradigm, it is essential to repeat a certain process for an uncertain number of times, to repeatedly transform entities until a certain property or condition is met; or even to iterate over lists (defined as parameters or attributes). As mentioned, recursive cycles are built by connecting an edge from a output port of a node to an input port of a parent node.

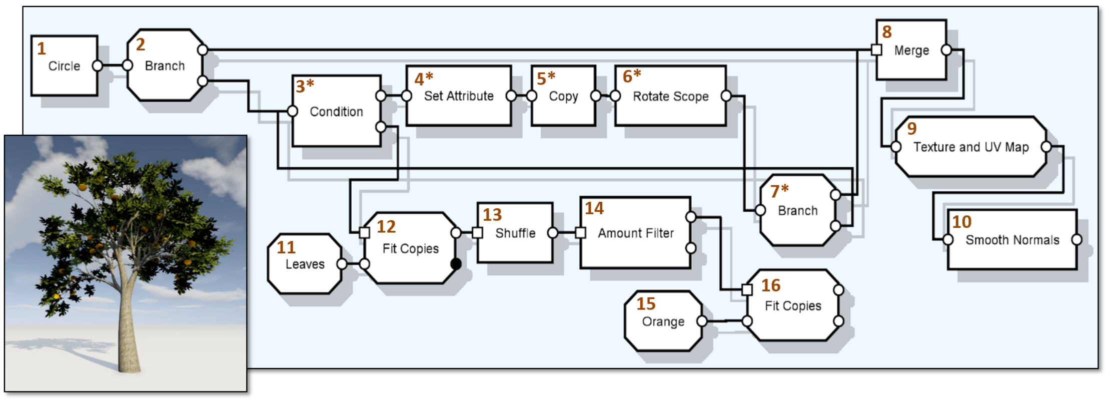


Figure 3.4: Cyclic node sequence in the generation of an orange fruit tree. The node numbering indicates the topological ordering of the nodes and the ones marked with an asterisk are bound to be executed multiple types, as they are found within a loop.

Topological ordering is only possible when graphs are acyclic. This means that, for recursive graphs, cycles have to be identified and some edges hidden from the topological sorting algorithm (see Figure 3.4). The algorithm to find such edges proceeds as follows:

1. Mark all edges as ’unvisited’ and ’non-cyclic’;
2. For each graph node, perform a depth-first graph iteration starting with that node N, each time with an empty list of visited nodes V:
   (a) Add N to V;
   (b) For each ’unvisited’ outgoing edge of that node, check the destination node M.
      i. If M is contained in the list of visited nodes, mark the edge as ’cyclic’.
      ii. Otherwise, mark the edge as ’visited’, and execute recursively at a) with M and V.

After these steps the process can proceed as before, with the single exception that, for the topological sorting, the edges marked as ’cyclic’ are not considered in the equation.

However, during the graph execution, all edges are considered. Nodes connected to an output port of another node are added all the same to Q even if the edge connecting them is cyclic. Again, the addition to the queue should follow the topological order.

3.3.2 Impulses

Unlike other nodes that feature input ports, source nodes are executed only once per graph. Multiple executions can only be achieved by having entities triggering the nodes.

Source nodes have the possibility to create a virtual single input port, of the generic ’Entity’ type. Entities queued at this input port are, in fact, discarded, but serve the purpose to trigger the node execution. This way, multiple node executions can be triggered if several entities are sent to the queue. These entities are called *impulse* entities, because they are semantically void and serve no purpose but to create an impulse that forces the node execution.

They can also be used to carry attribute data and then be accessed and used for building expressions (see Section 3.4.1). Entities created by the source through impulse influence are considered descendants of those impulses. As such, the attribute data is also passed from the impulse entities to the descendants. This is exemplified in Figure 3.5.

Another use of impulse entities is to supply ports that are optional. Since nodes requires all ports to be filled in order to be executed, impulse entities are the means to guarantee the execution (similar to passing a *null* reference in many programming languages). Optional ports are neither identified nor marked in a particular way - it is up to the node itself to establish internally what kind of input data is mandatory.

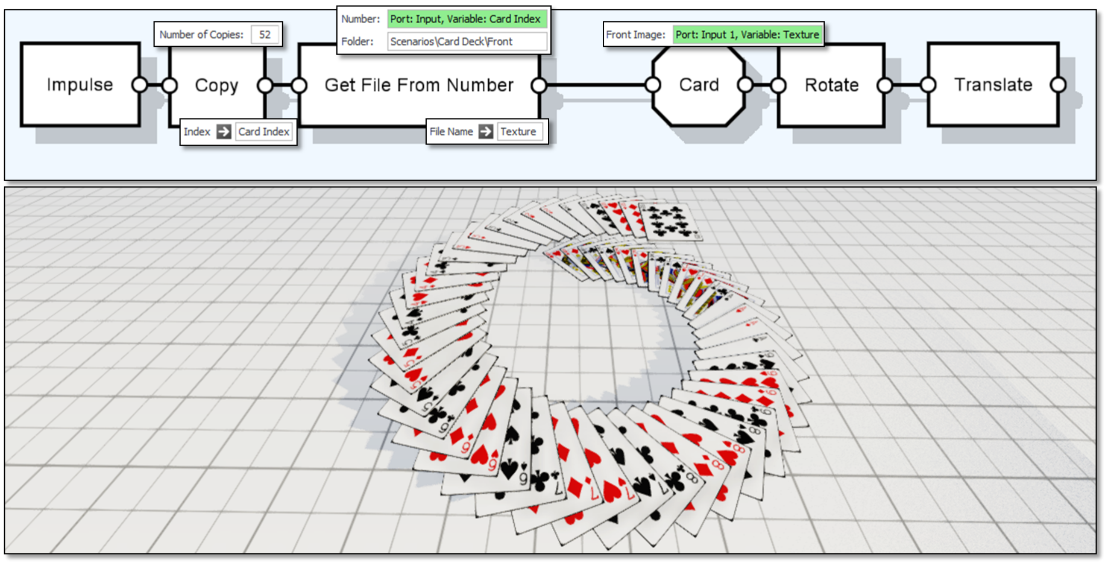

Number of Copies: 52
Number: Port: Input, Variable: Card Index
Folder: Scenarios\Card Deck\Front
Front Image: Port: Input 1, Variable: Texture
Impulse
Copy
Get File From Number
Card
Rotate
Translate
Index → Card Index
File Name → Texture

Figure 3.5: Example of impulse application in Procedural Content Graphs. The "Card" node is a source node that returns the model of a playing card, given the indication of the texture to apply. To force the generation of a full playing deck, an impulse is created and copied for the 52 cards, where the "Get File From Number" node uses the card index to fetch a different texture for each card.

3.3.3 Synchronization

Procedures can also be in charge of performing IO operations, either to the disk (files, databases. images, etc), or to remote location (web sites, web services, databases) or even sensors and peripherals (accelerometer, camera, mouse, keyboard...). The answer to such IO operations is often asynchronous, yet the execution of PCGRs is synchronous, meaning that each node has to be finished before it proceeds to the next.

Ideally, such synchronization should take place within each node. For instance, a procedure that accesses the webcam to obtain data should block until the image or video has been captured and converted to a entity that can be returned to the graph flow.

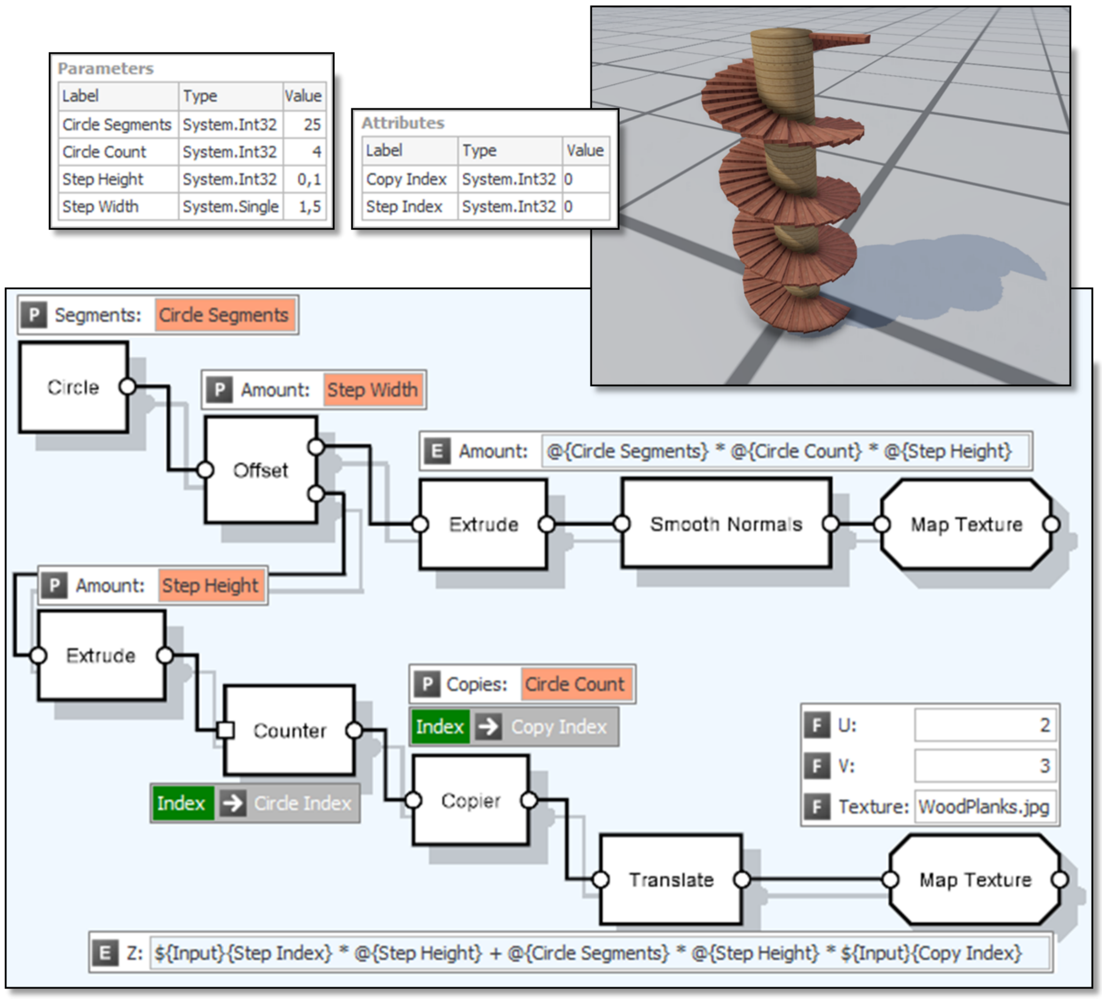
Another possibility is to allow graphs themselves to use nodes that send asynchronous requests, issuing impulse entities just so the remaining graph execution can proceed. This way, other procedures can be executed, optimizing the process. Later nodes can then hold, wait or simply perform a non-blocking verification, forcing a change in the flow until the response is received.

3.4 Aspects

Procedures/nodes are the basic blocks which perform entity instantiation, loading, transformation, or analysis on the input entities. They can also be used to perform any other kind of instruction, which may not be dependent specifically on the input entities (such as take a screenshot, connect to a web service or send a message to an external program).

Next to the entities that are passed to the nodes through their input ports, a procedure can depend on *node parameters* and *node attributes*, generically called *node aspects* (see Figure 3.6). They constitute the secondary type of data that is manipulated within graphs. They are not the actual content to be produced, but are essential to control the flow of entities and behavior of graphs and procedures. For instance, they can be used to define how much a rectangle is extruded, the noise frequency that a terrain should have, or the text of a message to be printed on a console.

Like procedures, graphs contain their own set of *graph aspects* and can be accessed globally inside the graph. As in many strongly typed languages, their existence, name, type and initial value must be declared before they can be used. They are defined as graph properties, together with the remaining metadata (name, description, etc.). As a result, their use is bound to the scope of that procedure or graph.



Parameters

Label	Type	Value
Circle Segments	System.Int32	25
Circle Count	System.Int32	4
Step Height	System.Int32	0,1
Step Width	System.Single	1,5

Attributes

Label	Type	Value
Copy Index	System.Int32	0
Step Index	System.Int32	0

P Segments: Circle Segments
Circle
P Amount: Step Width
Offset
E Amount: @{Circle Segments} * @{Circle Count} * @{Step Height}
Extrude
Smooth Normals
Map Texture
P Amount: Step Height
Extrude
Counter
Index → Circle Index
P Copies: Circle Count
Index → Copy Index
Copier
F U: 2
F V: 3
F Texture: WoodPlanks.jpg
Translate
Map Texture
E Z: ${Input}{Step Index} * @{Step Height} + @{Circle Segments} * @{Step Height} * ${Input}{Copy Index}

Figure 3.6: Construction of a stairway via attributes. An offset is applied to a circle to extrude polygonal steps. These are counted and receive an index attribute. The steps are copied and an additional copy index is added. The steps' translation is steered via these attributes and node parameters (white indicates constant values ("F"), red refers to graph parameters ("P"), while light blue represents expressions ("E")). Arrows indicate attribute forwarding.

The difference between a parameter and an attribute is that the first is a property of graphs and nodes, while the latter is a property of entities (see Section 3.2). For each declared parameter, one single copy of its value exists within the graph. As such, they are especially useful as centralized constant definitions that are referenced multiple types throughout the graph.

On the other hand, for each declared attribute, there will be an independent value stored in the hash map of each entity that flows within the graph. The attribute declaration works as a key for the entity hash map. In practice, an explicit storage of the value is not required if the value matches the default value of the attribute. Simply, if the search in the hash table fails, the default attribute value is used. In this way, all entities in the graph can be considered to be always carrying all defined attributes.

When a node is instantiated, node parameters are initialized with a certain default value, but can be changed either by assigning another fixed value (a numeric value, a character string) or expressions. These, in turn, might include arithmetic operations, external pure function calls (e.g. sin(x), cos(x), stringlength(str), etc.) or refer to any graph aspect. Node parameters can also have assigned constraints, which help ensuring its correct application. For instance, a parameter that defines a relative size can be restricted to a floating-point value between 0 and 1, while a parameter that indicates a name can have a character limit and be checked towards a certain regular expression.

Control attributes, on the other hand, are operated by *linkage*. They cannot be assigned fixed values or expressions, but only be connected to a graph attribute (or rather, to its key), which is expected to have a determining value assigned to it. An example of application is the 'Order By' collective node, which receives a list of entities to sort by generic criteria, defined as attributes. By passing the link to the graph attribute, the individual entity values can be read and processed in the sorting algorithm.

Both parameter and attribute values are mutable, although the method to assign new values are different and so are the implications. Changing parameter values by effect of an entity alters what can be seen as the global graph state. As a result, this could lead to unforeseen side effects for other entities and therefore should be handled with care. Setting parameters is done using *reflection nodes* (see Section 3.7.3).

Attribute change, on the other hand, involves a local change, since it occurs per entity. Such a change is easier to track, understand and control. The assignment of attributes can be done via procedures, following the concept of *data forwarding*: as entities flow through nodes or even specific ports, they can be assigned different values to the attributes via their hash tables. For example, a 'Counter' node, which takes a list of entities as an input, is able to assign the total number of entities and a unique index for each one to the passing entities (see Figure 3.6).

Although at a first glance it might seem cumbersome to provide attribute definitions for an entire graph, there are several important advantages to it. First, the user keeps an overview of all attributes that entities carry, which is often convenient. Second, it is easier to optimize memory usage, because most intermediate values that nodes could attach to entities might never be used afterwards. Third, it releases the need for namespace handling, as attributes are merely copied to the current graph scope.

3.4.1 Building Expressions

Indicating a fixed value to a node parameter will result in the same value being used across all node executions. As mentioned, the alternative is to resort to expression, where more complex calculations can be performed based on the input data and references to graph aspects.

As mention in Section 3.3 procedures are executed throughout several rounds, until the queues of all input ports are depleted. Expressions are evaluated before each round is executed, after the entities to be processed in that round have been identified. The notation to refer to an input entity is:

${port_name}

where *port_name* is the name of the port from which the data has arrived. If the port is a collective port, the same reference returns the first item of the list.

To refer to a graph parameter, the notation is:

@{parameter_name}

where *parameter_name* is the name of the parameter declared in the graph. On the other hand, to refer to a graph attribute value, the notation is:

${port_name}{attribute_name}

where *${port_name}* refers to the entity from a certain port, as before, and *attribute_name* refers to the name of the attribute. In other words, it returns the value of the attribute for the indicated entity.

Normal arithmetic operators can be used, such as:

*(@{parameter_name} + 5) * 2*

If a given type has subproperties or subfunctions, they can be access via the dot (.) operator, such as:

${port_name}{direction}.X + ${port_name}{direction}.Length()

In addition, expressions may incorporate static functions to perform mathematical or algebrical operations, such as:

Math.Min(@{parameter_name}, 100) or *Vector3D.New(2,4,5)*

Finally, they can also include quick static functions that operate over specific content types to perform quick consultations (e.g. to calculate the whole surface area of a shape or simply to verify if an entity is of that specific type):

Shape.Area(${port_name})* or *Shape.IsShapeEntity(${port_name})

As in any programming language, type compatibility is essential for a successful expression evaluation. Strings cannot be divided by integer numbers, if such operation has not been defined, an error should be expected. As for the final result, implicit conversions can be attempted, as the expression type and the declared parameter type are known and therefore parsing or conversion between the two can be attempted (e.g. integer to string, double to integer).

3.4.2 Attribute Control

In the process of controlling several entities, the consultation, interchange and aggregation of property information among entities are required to achieve a certain level of inter-operation and context-based development. For instance, when generating a building, it might be necessary to access and store the name of the adjacent streets. Or, when focusing on streets, the indication of which neighborhood it belongs to may be required to define the road state. Or, when developing neighborhoods, the city center location may need to be calculated from their sub-parts, so as to distribute points of interest. This is always achieved by first storing such properties into attributes, which offer the necessary flexibility to access, modify and share information among entities for the single purpose of controlling the generation process itself.

Attribute manipulation operations can assume different natures, which can be categorized as follows:

- **Extraction:** Load an intrinsic property from an entity into an attribute. This can range from simple fetchers (e.g. return the name of a street or the scope of the shape) to more complex searches (e.g. determining the maximum height of a surface) or calculations (e.g. the total area of a shape).

- **Simple Assignment:** to store a fixed value or the evaluation of an expression into an attribute of a given entity. This way, calculations can be performed on graph parameters and attribute references using mathematical or other aspect-based functions (see Section 3.4.1). Also, this introduces the possibility to copy, rename or swap attributes within the same entity.

- **Multi Assignment:** Same as the previous case, except that, by featuring several input ports, attribute values from specific entities incoming from different ports can be referenced in the expressions. Sharing, swapping or moving values can therefore be processed among attributes of different entities. It also allows for more advanced calculations to be performed.

- **Aggregation**: By collecting entities using collective ports, aggregation functions can be applied on sets of attribute values, such as minimum, maximum, average, sum and count.

- **Maneuvering:** When dealing with entities containing other entities, attributes values can be copied from the parents to the children, vice-versa, and between children of different types, if applicable. For instance, one might need to copy the color information from the faces to the vertices, before these are changed and then passed to adjacent edges. It is important to notice that the maneuver action from an attribute value from a child to a parent might require some selection or aggregation in cases of many-to-one relationships.

The other means of data control is through the already mentioned *attribute inheritance* from ancestors to their descendants. It differs from the previous cases, as it is a characteristic of graph execution and is not explicitly handled. It is, however, a fundamental feature, as it avoids losing data during the generation process, which can rely heavily on entity splitting and reassembly.

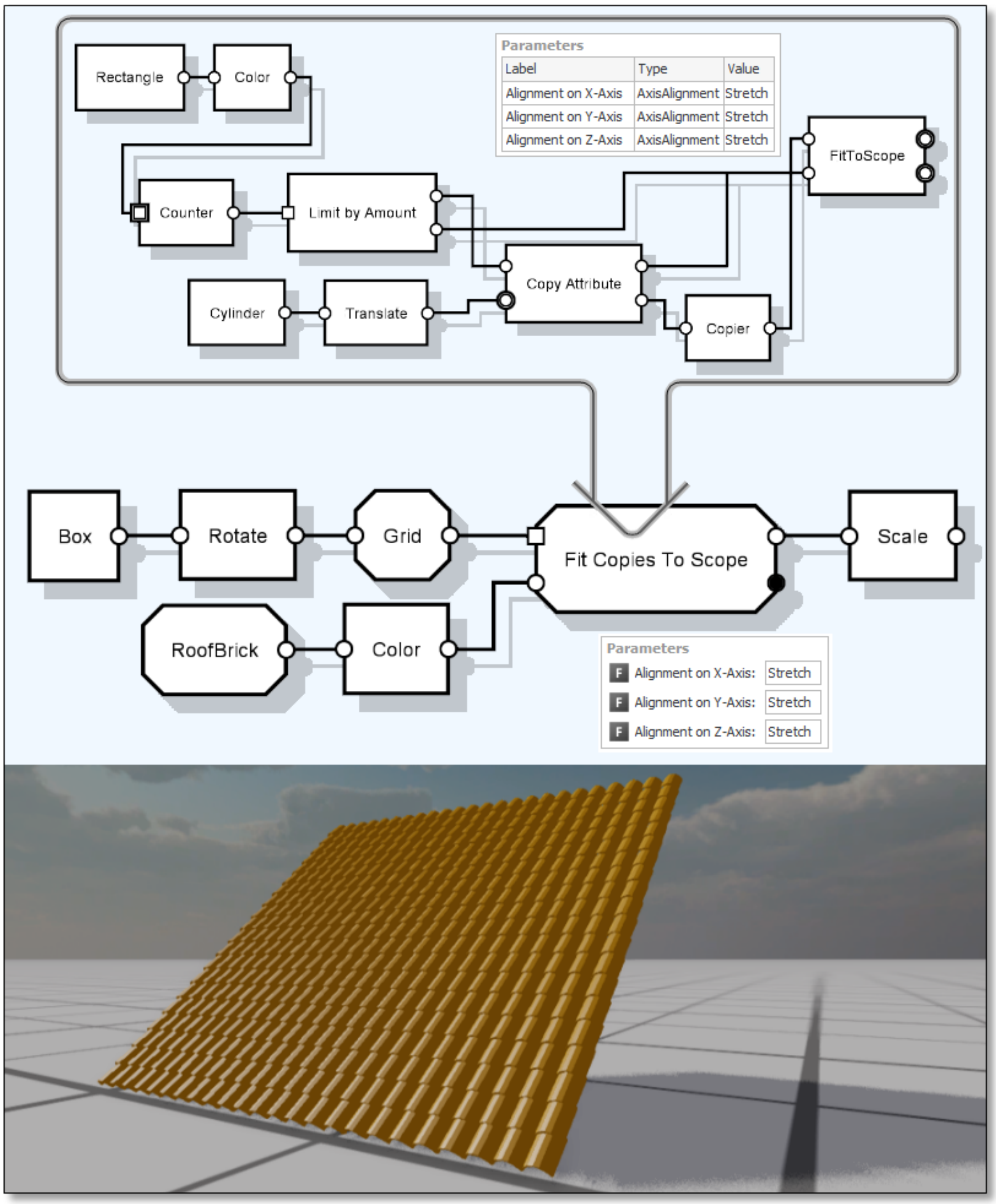
3.5 Encapsulation

Just like procedures, graphs execute series of program instructions that operate on entity data, behaving according to parameters and attributes. In practice, a graph is a procedure and, as such, can be materialized as a node, just like any other. This process is called *encapsulation*.

The operation to transform a graph G into a new node, N, ready to be used in a supergraph S is relatively straightforward (see Figure 3.7). Any input/output port of a node in G can be given a more meaningful label and marked as a *gate*. A gate will serve as a port when this graph is used as a node in a supergraph. Graph parameters of G become node parameters of N by default, yet they can be hidden, if so desired. The same applies to attribute definitions in G. Hereby, the attributes in S can be transferred to G and/or forwarded back from G to the scope of S.

Regarding execution, a subgraph will always be executed as long as possible and, only when its execution queues are empty, will the supergraph continue its execution. As a result, this complete local execution of encapsulated graphs can also be helpful to control the execution order, in cases where this is needed. Once the encapsulated graph finishes execution, all attributes that have been defined within this subgraph (but not those passed via data forwarding) are removed from all entities before proceeding to the supergraph.

An important feature is the difference between executing a standalone graph and an encapsulated graph. As mentioned, any port of a graph can be set as a gate by the user. When encapsulated, a gate receives its data from the supergraph, instead of from within the subgraph. In Figure 3.7, the two gates are preceded by series of nodes (rectangle, color, cylinder and translate) that are considered only when the graph is executed independently, which is useful for tests. When encapsulated, the graph will ignore these nodes and consider only the input from the supergraph.



Parameters

Label	Type	Value
Alignment on X-Axis	AxisAlignment	Stretch
Alignment on Y-Axis	AxisAlignment	Stretch
Alignment on Z-Axis	AxisAlignment	Stretch

Parameters

F	Alignment on X-Axis:	Stretch
F	Alignment on Y-Axis:	Stretch
F	Alignment on Z-Axis:	Stretch

Figure 3.7: Encapsulation of a graph that extends the 'Fit-to-Scope' operation to lists. The thick-line ports indicate the gates of the graph, which are mapped to the encapsulated node's ports. Graph parameters are also mapped to the node's parameters. The supergraph uses this encapsulated graph to easily copy and fit a brick rooftop into each cell of a rectangular grid (both brick and grid are also encapsulations). The result is a detailed roof surface.

Encapsulating a graph and properly organizing its parameters, attributes, and ports to define more complex operations is a well-known concept of *component-based development* in software engineering practices. One of the main advantages is reusability - users can assemble operations in a way that the encapsulated node can be reused in many situations and shared with other users. As long as one is aware of its usage, it is not necessary to know about its internal functioning - the operations become a black box.

Encapsulation also allows the definition of more complex structures for a higher degree of semantic manipulation. By successively encapsulating graphs, one can achieve a higher level of control where the manipulated nodes can represent increasingly complex architectural structures (pillars, windows, balconies, doors...) instead of low-level operations.

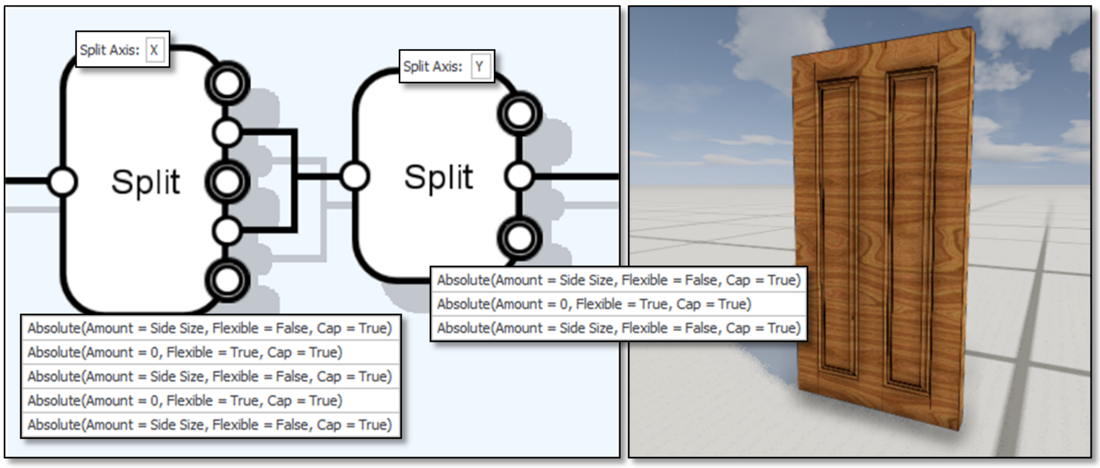
3.6 Augmentations

By default, procedures have a static *signature*, i.e. they offer a fixed set of available node parameters, attributes and ports. Yet, an operation such as the split, which cuts polygons, requires the enumeration of several slices and the customization of each one. In CGA Shape grammars, each slice carries information about size, flexibility and output rule name. Only by having all such information at once can the procedure pre-calculate the remaining available splitting size and adjust each slice size accordingly.

The generic way for PCGRs to deal with such flexible design annotation consists of *augmentations*, which can be seen as an extension to nodes. An augmentation is a structure that aggregates parameters, attributes or ports. Again, for a split node example, an augmentation is the structure that holds the information about the slice and contains the output port where to send the result. Any number of such augmentations can be added, and doing so affects the number of output ports of the node.

Generally speaking, augmentations are useful to indicate *lists* of constraints or guidelines that a node should consider in its operation. In most cases, the order of the listed augmentations defines priorities in the node configuration and changes may yield different results. Consider the following examples where augmentations are especially useful:

- **Merging and Grouping**: enumerate criteria for merging and grouping, e.g. based on attributes of the entity (size, material...);
- **Ordering**: enumerate criteria for sort the records in a descending or ascending order, e.g. based on attributes of the entity (size, material...);
- **Splitting/Decomposition**: define the size/criteria of split slices and provide additional ports to output them (see Figure 3.8);
- **Stochastic/Conditional decision**: decide upon control flow according to sets of probabilistic or attribute-based conditions;
- **Creation**: enumerate vertices of a polygon (or mesh) to be assembled;
- **Custom Import/Export**: load/save custom attributes that may be associated to the geometric data from/to a database, e.g., geospatial data.



Split Axis: X

Absolute(Amount = Side Size, Flexible = False, Cap = True)
Absolute(Amount = 0, Flexible = True, Cap = True)
Absolute(Amount = Side Size, Flexible = False, Cap = True)
Absolute(Amount = 0, Flexible = True, Cap = True)
Absolute(Amount = Side Size, Flexible = False, Cap = True)

Split Axis: Y

Absolute(Amount = Side Size, Flexible = False, Cap = True)
Absolute(Amount = 0, Flexible = True, Cap = True)
Absolute(Amount = Side Size, Flexible = False, Cap = True)

Figure 3.8: Application of two split augmentation nodes to create detail on a wooden door. To each split augmentation, there is an associated output port that is added to the node design.

An important observation is that augmentations cannot necessarily be mapped to a simple sequence of nodes or encapsulation. This is the case for operations that require some kind of pre-calculation (e.g. split), evaluation (e.g. condition) or any other "atomic" organization (e.g. grouping or sorting).

3.7 Imperative Paradigm

As in content generation grammars and other dataflow programming languages, PCGRs would follow a rather functional paradigm, avoiding state and mutable data. As advantageous as it may be, often data management can be facilitated if imperative-language features are introduced, such as the control over entity references and the manipulation of procedure and graph states. This section will review the mechanisms that make such control possible.

3.7.1 Reference Management

As entities are objects, they are identified uniquely and can be manipulated by merely passing and accessing its reference. When an entity is passed to a node, its content is transformed, but the reference to the container remains the same.

Unless copied (in divergent ports or inside specific nodes), the entity reference that goes *in* equals the one that goes *out*. That being said, it is possible to keep the reference to an entity at a certain point of the graph so as to be able to easily access it at a later point using only port and edge connections.

In most grammar-based approaches, a great issue that prevents a safe control over entity references is the lack of an explicit derivation order. In fact, one of the main advantages of such approaches is that they can be easily parallelized, as the manipulated objects are independent copies, not references. On the other hand, it prevents the use of references, as no predictable behavior exists and concurrent accesses may occur.

Procedural Content Graphs are executed in sequence, one node at the time, and the 2 rules mentioned in section 3.3 provide a rather clear view on its order. That being said, reference control becomes possible.

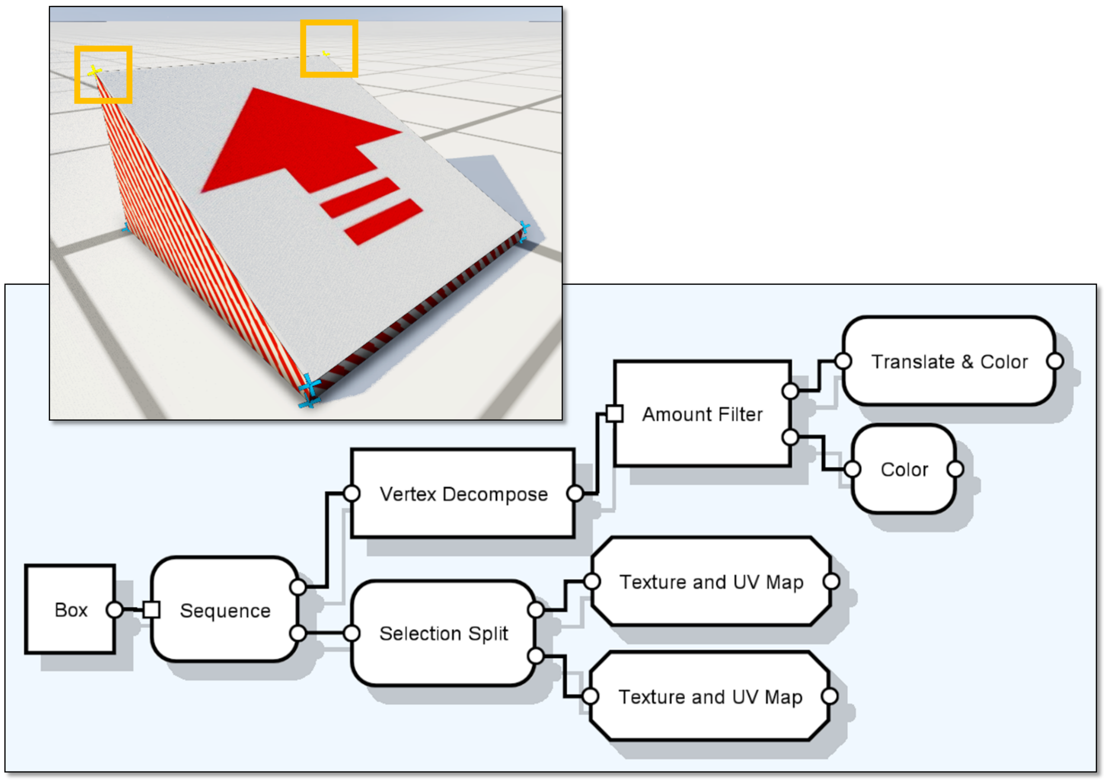


Figure 3.9: Illustration of reference management application to create a ramp. The operation consists in selecting 2 vertices (highlighted in yellow) and translating them, without splitting the mesh. The "Sequence" node aids in the definition of an execution order.

An example of the benefits of reference access is portrayed in Figure 3.9. Considering that a mesh is composed by a tight mesh of vertices, edges and faces, there is no way to access and alter a property of an entity vertex without decomposing the mesh into separate entities as well. But a decomposition into independent entity copy would not propagate the change onto the original mesh. The only way would be to reassemble the mesh with the original connections, an action that would be very tiresome, if not impossible to achieve. Using references, the mesh can be decomposed into the vertices, which are altered by the first row of nodes. The ordering rule guarantees that the reference to the mesh is held for the second row of nodes. This manipulation is facilitated by nodes such as the "Sequence" node, whose augmented nature allows the flexible addition of output ports that simply forward copies or references the entities incoming in their input ports.

3.7.2 State Persistence

In a functional paradigm, grammar procedures are usually seen as pure functions, which do not feature any internal state and whose result is only dependent on the arguments. Hereby, side effects are avoided. However, keeping an internal state, when properly employed, may be beneficial to achieve a certain degree of control and performance (e.g. in object-oriented languages).

In a procedural content graph, the nodes are objects that can include, besides the definition of ports, parameters and attributes, several additional internal properties, as well as initialization functions, called at various moments of the graph execution (e.g. when the graph starts or when the node is first executed). Such properties can be used to define an internal state of the procedure. Like functions, procedures are labeled as impure or pure if they do or do not persist an internal state, respectively.

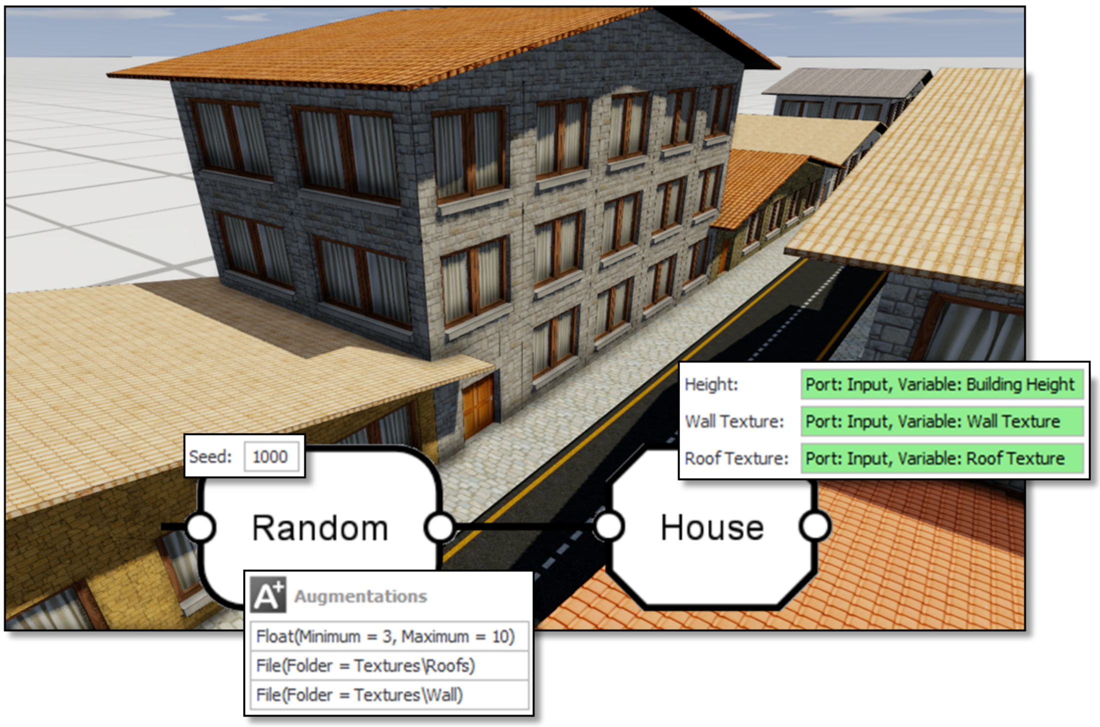

Seed: 1000
Random
House
Augmentations
Float(Minimum = 3, Maximum = 10)
File(Folder = Textures\Roofs)
File(Folder = Textures\Wall)
Height: Port: Input, Variable: Building Height
Wall Texture: Port: Input, Variable: Wall Texture
Roof Texture: Port: Input, Variable: Roof Texture

Figure 3.10: Use of the random node which features an internal state to return random height values and textures.

Examples of application include:

- Keeping track of the total number of processed entities (throughout all calls of the procedure), which can be useful to create unique identifiers;
- Holding single instances of IO stream handlers (e.g. file, databases, web services, sensors), improving performance and memory consumption;
- Holding the state of a random number generator, so as to return different values for

subsequent calls;

- Maintaining a cache so that heavy function or procedure calls can be spared, for identical inputs.

Figure 3.10 shows how the "Random" node is used to obtain random building height values and random building textures for both roof and walls. The node is executed per building lot and, although the node run independently for each one, the random generator state is kept so as to return different values across procedure calls.

A seed value is used to configure the generator, but, unless it is changed, the generator is not reset. The fact that the seed can be manipulated is essential to ensure *graph determinism*, i.e., that for the same input (as composed by input entities, parameters and attributes), it exhibits the same behavior across several executions. In other words, although the final result is the outcome of randomness, this randomness was obtained always in the same way.

3.7.3 Reflection Nodes

The concept of *reflection* refers to the ability of a programming language to perform analysis and manipulation of the designed structure and behavior of the program at runtime. This includes access and modification of values, functions, types and other metadata. In PCGRs, the same is possible using *reflection nodes*, which have the ability to operate over graph, edge and node details.

One of the main uses of such nodes is *parameter mutation*, as mentioned in section 3.4. As parameters are properties of graphs and described using unique identifiers, such nodes can simply access them by their key and alter their values. This allows a powerful control over the graph global state, which can even persist throughout several executions. Evidently, such potential requires equal caution, as modifying the global state can result in unexpected side effects.

3.8 Dynamic Loading

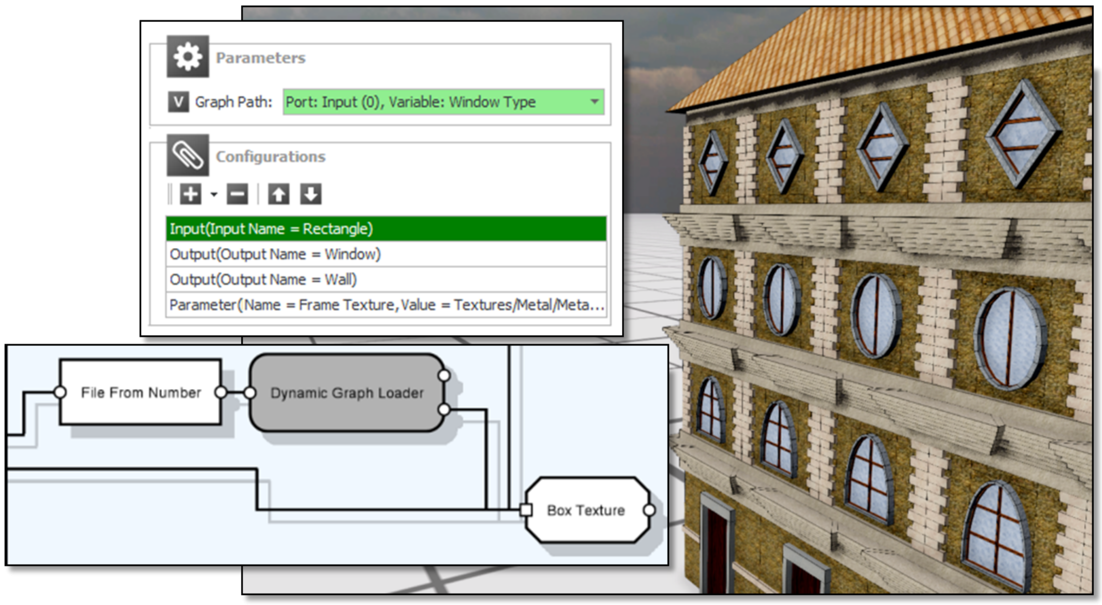
Dynamic loading in PCGRs is the process of deciding, loading and processing nodes during graph execution by means of *surrogate nodes*. These nodes have a parameter that indicate the name (or path) of the *original node* to load (either a standard or encapsulated procedure), defined as a string that can, for instance, be mapped to a global parameter or expression. As a result, they can assume different values for each entity or situation. The node execution can then be processed under specific conditions, a specific number of times, with particular variations.

Surrogate nodes can use augmentations to define the signature of the node: parameters, attributes, inputs and outputs, along with their names and types. After these have been set up, they will fit into the graph as any other, in both connectivity and parametrization and it will be able to dynamically load any node that follows the defined signature. During execution, the name of the original node to load will be evaluated and the node instantiated. The parameters, attributes and input data are then mapped from the surrogate node to the created node. After being executed, its output data of the latter is again mapped back to the surrogate node, which forwards the data to the containing graph.

In other words, this feature is equivalent to delegates, reflection or dynamic method calls, which are features present in some programming languages. The purpose of dynamic loading is to provide graph development not only with more flexibility and extensibility, but also with increased manageability options, which will be explained in the next sections.

3.8.1 Exception Encoding

Visual programming languages, as well as grammars, tend to be somewhat strict in their specifications. The design of a house, from the starting footprint to the details of its window ornaments, follows a rather inflexible definition. For instance, if one would like to keep the building and facade layout, but introduce the choice of a completely different window style, it would require adding an exception to the main ruleset (or graph). Managing encapsulated nodes can greatly ease such a task, but the need for an explicit encoding of each exception is not convenient.



Graph Path: Port: Input (0), Variable: Window Type

Input(Input Name = Rectangle)
Output(Output Name = Window)
Output(Output Name = Wall)
Parameter(Name = Frame Texture, Value = Textures/Metal/Meta...

Figure 3.11: Use of a dynamic loading node, configured to load graphs that receive a rectangular wall shape as input and output the window and the remaining wall. In this example, the floor number is first used to obtain a file path of a graph from a folder and save it to an attribute, resulting in a different choice of window for each floor. Alternatively, the choice of window could also be set, for instance, as a parameter of the building graph.

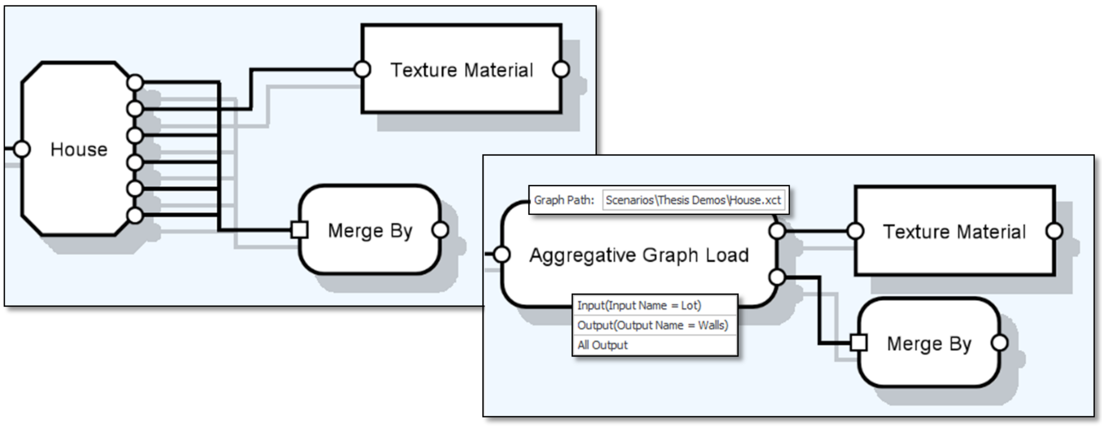
A method to prevent such a rule explosion in grammars has been presented in [KPK10] through abstract structure templates - a non-terminal rule signature, which could then be instantiated and passed as parameter to be used from other rules. In PCGRs, surrogate nodes can attain the same result. The indication of the node name and respective parameters are passed through the graph's parameters, and all that it takes is having the surrogate folder have its augmentations configured so as to map the parameters and link the input and output ports to the remaining nodes.

The example in Figure 3.11 goes even further in exception encoding and extensibility, as the graph name is extracted randomly from a given folder. This means that the graph containing the surrogate node can be extended without having its structure directly modified. All that is necessary is for new graphs, following the same signature, to be placed within that folder.

3.8.2 Port Aggregation

An issue that is tied to the use of encapsulation is the difficulty to distinguish the various parts of the generated structure. Using several output ports to attain such differentiation is the most common and recommended path. For instance, a graph that generates a building out of a lot might return walls, chimney, windows etc. using several output ports, which is useful if they are to be further developed outside the encapsulated graph.

However, often this results in very large nodes featuring a too large number of ports, which quickly becomes impractical to use, since all of them have to be connected to the following nodes. In fact, this distinction may not be necessary on every occasion and it would be simpler to just have them merged together for easier manipulation. This middle ground between having few or many output ports is achieved using the *aggregative* variant of surrogate nodes.


Graph Path: Scenarios\Thesis Demos\House.xct

Input(Input Name = Lot)
Output(Output Name = Walls)
All Output

Figure 3.12: Dynamic Loading Node used to aggregate ports of a given graph node, therefore reducing the number of output ports that need to be connected to the "Merge By" node.

Instead of adding the encapsulated node itself to the graph, the surrogate node that will load it contains a particular kind of augmentation that, instead of performing a one-to-one mapping of output ports, is able to congregate all the data from all non-listed ports. This is displayed in Figure 3.12, where only the building wall are differentiated, while the rest is aggregated into a single "All" output.

3.8.3 For Loop

Repeating a certain sequence of operations is typically achieved using recursive connections, as mentioned in Section 3.3.1: entities are sent back to an already traversed node, an action that will be repeated unless the entities are diverted to a different path through a node that imposes some sort of exit condition. This is especially useful when the exact number of loops is not known beforehand. Otherwise, many programming languages offer a more practical, iterative alternative - the *for loop*.

In PCGRs, the for loop statement is emulated though a node with the same name that accepts an "index" attribute, used to decide if an incoming entity is sent to the first or second port of the node, before the attribute value is incremented. In other words, the index is a value stored within each entity, contrary to the usual operation of the language statement, where this index typically is independent of the handled data. While this approach works for single entities, it is sometimes cumbersome to use and even more complex if the exit condition of one entity has to affect the flow of other entities.

This type of programming statement is achievable with dynamic loading approaches: the "Iterative" surrogate node, the original node is loaded and then executed a certain number of times, as indicated by a parameter. However, unless the original node is a source node, in practice it will only be executed once, as the input ports are not restocked. Therefore, the surrogate node features a specific type of augmentation where a link between an output port and an input port of the original node can be designed. This guarantees that the input ports are filled and the process can indeed be executed several times.

It is possible for the original nodes to operate based on information of the current iteration index, as it is common in the "for loop" statement. Doing so simply requires the original node to include a parameter of integer type that the surrogate node can access. Its name should then be indicated in the "Iterative" surrogate node configuration, which uses this information to update the corresponding parameter of the original node with the current iteration index.

3.8.4 Debugging

Debugging is another function that can be performed dynamically with the aid of specific surrogate nodes. This introduces quick and detailed reporting abilities on the referenced node's performance and data management process: execution time, amounts and types of processed data, actual expression evaluations, among others. These values are orderly logged in the available standard output. Comparatively, performing "manual" logging operations inside the procedure or graph would be too cumbersome to achieve. On the other hand, while relying on external development tools (see Section 5.2.4) can be of equal usefulness, their reports may not be transferable to other environments where the graphs are executed.

Generally speaking, the execution of graphs and procedures include several moments which can be interrupted in the sense of obtaining important details about their present state. This includes the moments before and after each procedure execution and, on a smaller scale, the moments before and after each round execution. Debugging is one of the custom actions that can be performed at these occasions, but doing so requires the execution of a node with a specific *execution modifier*, which invokes these actions. Assigning this modifier to an encapsulated graph guarantees its propagation to the subnodes and, recursively, subgraphs. Consequently, and using reflection abilities (see Section 3.7.3), many reporting details can be obtained, including:

- Actual node sequence and data flow details;
- Quantitative and qualitative (e.g. visual) snapshots of the result at particular moments;
- Relative and absolute time spent by each operation;
- Relative and absolute data produced at each node.

The understanding of possible graph flaws, inefficiencies and erroneous behaviors is the key for their correction, which makes the usage of such surrogate nodes essential.

3.8.5 Parallelization

In processes as time-consuming and intensive as the ones found in massive content generation, the possibility to parallelize operations can result in great performance optimizations. The derivation of grammars such as CGA [M¨+06] is very flexible and, given that they operate over independent entity copies, the whole generation process can occur simultaneously, in a transparent way, i.e. without the need for user intervention.

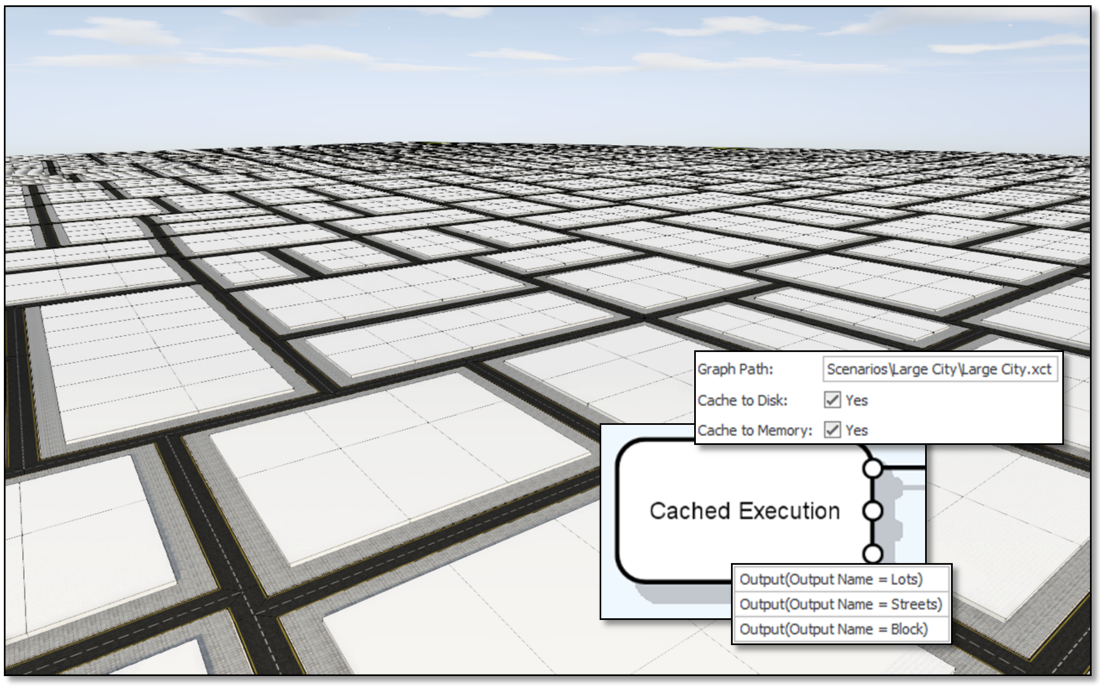
Parallelization is PCGRs is more complicated to achieve for several reasons, the main one being the need for collective operations (whose applications will be better discussed in chapter 4) that must collect all preceding entities, requiring a convergence of parallel tasks. Adding to this is the need to guarantee an execution order for safe reference management (see Section 3.7.1) and the possibility for state persistence (see Section 3.7.2), which would make parallel execution too chaotic and unpredictable.

However, it is safe to assume that, for single nodes, the entities queued at the input ports can be handled independently and concurrently, as long as no relevant internal node state has to be maintained. It is on this consideration that dynamic loading can be effective: for the indicated original node, several copies of the procedure are created, each on a different working thread, to be executed concurrently. Afterwards, the input data, sent to the containing surrogate node is evenly split among the threads, the parameters evaluated and the procedures executed. In the end, the output data is extracted and sent to the surrogate's ports. Finally, all that is required is for the result entities to appear in the same order as they would as if handled in a single thread.

Any sequence of operations can be easily parallelized by simply encapsulating them and using a surrogate to refer to the node. It is important to underline that such a parallelization method is only compensatory for large amounts of entities or for heavier operations, otherwise the parallelization attempt might not only be pointless but also induce an unnecessary overhead.

3.8.6 Caching

The main idea behind caching is to optimize execution performance. Content that has been generated before and will be generated in the same way does not need to undergo the same generation process, especially if such a process is heavy and time-consuming. Dynamic loading can also be used to cache entity data, avoiding the re-execution of a given graph if no change has occurred.



Graph Path: Scenarios\Large City\Large City.xct
Cache to Disk: ✓ Yes
Cache to Memory: ✓ Yes

Output(Output Name = Lots)
Output(Output Name = Streets)
Output(Output Name = Block)

Figure 3.13: Optimization of the large city generation. The streets and lots are not regenerated every time thanks to the Cache node, which keeps a copy of the content, saving quite an amount of time.

Cache nodes receive the indication of the graph or node to be loaded and can choose to execute it dynamically or simply to output stored entity data (see Figure 3.13). Caching nodes is applicable to source nodes only, as caching operations based on input data would require complex equality and mapping verifications and require a heavy memory footprint. This is not an issue, though, since any sequence of nodes can simply be encapsulated to meet such a requirement.

Cache is discarded and renewed if the referred graph is changed, which can be verified by simply comparing the last modification date/time with the last execution date/time. Also, if the parameter values passed to the corresponding augmentation are changed, the cache has to be discarded and the graph executed again, as there is no way to predict the impact of the change to the final result.

Like other surrogate nodes, augmentations are used to define outputs, parameters and attributes, but no inputs, obviously. Parameters of the cache node itself serve as means to impose caching limits and to define caching locations. The caching process can be performed at memory level, which persists across executions within the same environment. It is fast, since the saved entities only have to be copied on each execution. Storing to disk is slower, yet persists across executions even within different environments.

3.9 Managing Semantics

As mentioned in chapter 2.2.5, the understanding and management of the semantics of objects being generated can help in providing a more powerful and higher-level control over the content. In the work of Tutenel et al. [KTB12], the semantics of objects to generate are listed, configured and maintained in a central semantic database. All such definitions are then used for several algorithms which employ such information to build virtual worlds consistently according to the rules, properties and expected behaviors of the involved objects.

In PCGRs, semantics is not stored and organized in such a centralized and explicit manner. Instead, semantics is present and specified in several ways, already mentioned throughout this chapter, but hereby summarized.

The first degree of semantics lies on the typing of manipulated entities (see Section 3.2), as well as their organization into a hierarchy of superclasses and subclasses, defining common characteristics and their relationships. Each entity has its own specific data model, which aggregates several properties and provides a higher-level knowledge about how they are structured and how they can be manipulated. As a result, more optimized and specific procedures can be built to handle this kind of knowledge. For example, using a grid-based vertex structure to define terrain surfaces is more appropriate than using a generic b-rep mesh representation, as the manipulation of this data becomes simpler, faster and more memory-efficient.

The second degree of semantics lies on the possibility of an entity to aggregate other entities, a possibility that can be recursively reproduced through grouping. This construction is useful to compose entities by their significance and decompose them when necessary. For instance, a house entity can aggregate facades, which in turn group windows and doors. Such grouping is useful to control buildings as a whole, but, if necessary, also allows an easy and structured access to the building subparts. The advantage of this approach is that the structure is flexible, meaning that it can be changed according to the needs.

The third degree of semantics lies on the definition of custom attributes in procedures and graphs, which add information to the manipulated entities inside their execution scope (facade number, floor number, object name...). This constitutes a generic method to build custom objects, according to manipulation needs. As procedures are able to read and write to external attributes as well, the same effect can be produced as having such properties as part of the entity structures.

Finally, the last degree of semantics lies on the encapsulation of sequences of rather generic procedures into more meaningful nodes. This is a very powerful feature, as it allows the definition of more complex structures, such as windows, balconies or houses. Eventually, design can be controlled by simply assembling high-level objects instead of low-level operations. In this sense, semantic control can lie more on the operations, rather than on the entities themselves and may work as effectively in generic objects.

3.10 Summary

The main contribution, presented in this chapter, consists of Procedural Content Graphs (PCGRs), a new visual language for the definition of procedural generation processes that addresses these issues through a graph-based representation. Here, nodes correspond to content manipulation procedures and the edges to data flow channels that connect to node ports, chaining these procedures. The generated content data is organized into sets of entities, which can be controlled through the use of parameters and aspects. Unlike other approaches, each node can feature several input and output ports, as well as handle lists and combinations of entities at a time, instead of single ones.

Management is aided by possibilities such as encapsulation, through which constructed graphs can be assembled into compound nodes that can be more easily reused, controlled and shared. Also, the concept of augmentation introduced additional design flexibility, by acting as node extensions that can be further configured. Dynamic loading, on the other hand, allows for procedures to be loaded by other surrogate nodes and executed under particular conditions. Unlike other node-based languages, PCGRs are not limited to a functional paradigm, but instead have a steady support of state persistence and its well-defined sequencing allows for safe entity reference management. All this characteristics contribute to a more powerful control over the semantics of the operations being executed and the content being designed.

4. Content Generation

Having explained the basic functional aspects of Procedural Content Graphs, this chapter will focus on demonstrating how they can be used to control the procedural generation processes and achieve a level of expressiveness, manageability and integration that was not possible before.

The chapter will start by an explanation of how this novel graph representation can be mapped to existing grammar approaches, in order to demonstrate that it is at least as resourceful and powerful. Only afterwards will the improvements regarding expressiveness and manageability be described, starting by enunciating new possible procedures such as merging, grouping, aggregation, amount filtering, sorting, combination, among others, and explaining how they can express unaddressed design ideas and manage content data in more powerful ways. Other difficult tasks, such as the incorporation of external data sources, the extrapolation levels of detail and the combination with constraint-based approaches will also be tackled.

A later section will then describe the possibilities of data integration, with a particular focus on the generation of textures and application-specific entities, explaining the benefits of its sequencing with the development of geometry and its usage to create challenges for virtual reality engines.

4.1 Mapping From Grammar Approaches

Content generation grammars, such as the ones presented by Wonka, Müller and Krecklau [KK11; KPK10; M+06; Won+03], are defined typically as a set of production rules *Predecessor* $\rightarrow$ *Successor*, where the predecessor is a non-terminal symbol and the successor is a set of one or more non-terminal or terminal symbols. Given a starting non-terminal symbol, the process consists of successively replacing symbols that match the predecessor with the ones indicated in the successor. For instance, in shape grammars, geometric shapes are the symbols that are created or successively transformed by means of operations.

In a procedural content graph, the same rule structure is represented by the graph connectivity, in the sense that entities (the non-terminal symbols) emanating from a given output port (the predecessor) flow through established edge connections to input ports of other nodes (the successor), which will further transform them. The possibility to create any number of connections between any two output-input port pair (of compatible types) induces that the same possibilities for rule convergence, divergence and recursion still apply. As for other grammar features, they can be easily reproduced as follows:

- **Parametric Rules** [M+06]: parameter passing is achieved by means of attribute handling (Section 3.4).
- **Conditional Rules** [M+06]: a condition node, featuring one input, two outputs and a boolean parameter/expression. If it evaluates to true, the entity received as input is sent to the first output, otherwise, to the second.
- **Scope Rules** [M+06]: pushing and popping scope states are achievable through an attribute of type List/Stack, which is written to and read from, just like any other attribute.
- **Split rules** [M+06]: a split is an augmented node, which allows a flexible definition of the split sizes and the introduction of a separate output port for each split. Snap shapes can also be introduced dynamically through specific augmentation types (Section 3.6).
- **Occlusion query tests** [M+06]: An occlusion query node uses its collective port to receive all the shapes, organize them into an octree and test for occlusions (within a certain distance, defined in the node parameters). The occluded and non-occluded shapes are returned via different output ports.
- **Generalization of non-terminal objects** [KPK10]: supported naturally through the typed nature of procedural content graphs (Section 3.2).
- **Connecting structures** [KK11]: supported naturally through the possibility to define multiple input ports or collective ports (Section 4.2.5).
- **Accessing and Creating Containers** [KK11]: supported by the merge and grouping abilities offered by collective ports (Section 4.2.1).

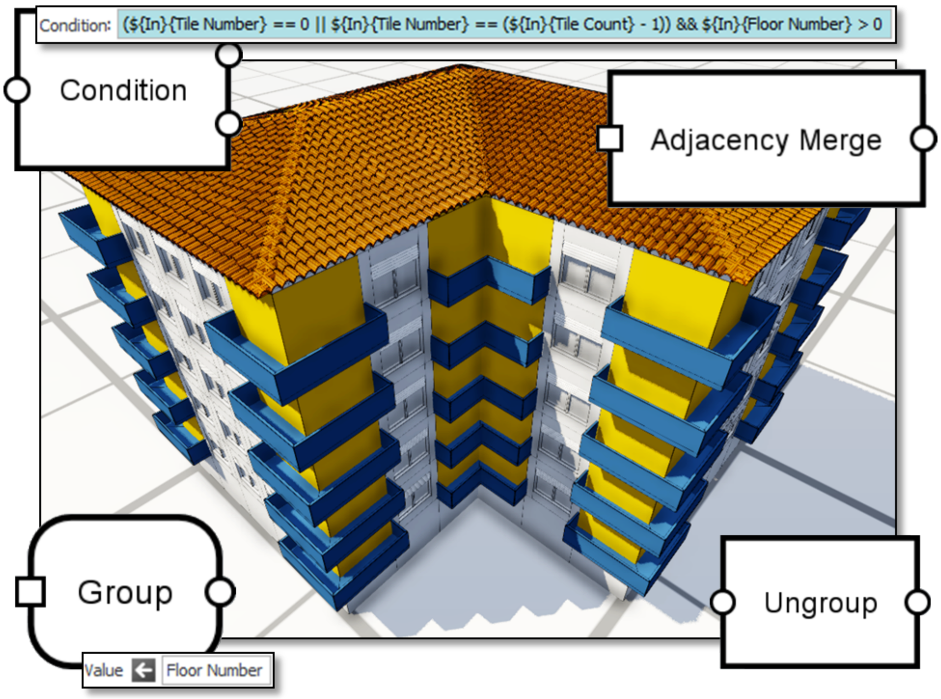
4.2 Improving Expressiveness and Manageability

In addition to being able to achieve the same designs as the mentioned grammar-based methodologies, PCGRs introduce enhanced expressive power, allowing the definition of specifications, rules and operations that were not possible before. Also, derived from their visual representation, PCGRs provide increased manageability features that facilitate the loading, manipulation and extrapolation of data. All these possibilities will be addressed in this section.

4.2.1 Merging, Grouping, Unification and Clustering

As mentioned, data is organized into entities that can be assembled or decomposed as manipulation needs arise. Shape grammar approaches use a top-down approach, transforming one simple entity into many complex ones, but there are many cases where joining back entities into single ones is not only convenient, but also necessary to achieve certain designs. This is not possible in such grammars, as they are only capable of picking and processing each entity individually. On the other hand, in PCGRs, this is possible through collective nodes, which provide the means to accumulate and congregate entities, so that they can be processed as a whole.

Hereby the concept of *grouping* refers to the process of collapsing the entities under a single container entity, according to certain criteria, such as spatial distribution, matching properties or common attributes. Grouping works primarily as a means to organize and structure entities, building an hierarchy, but which can be ungrouped an any point, retaining the original entity independence.


Condition: (${In}{Tile Number} == 0 || ${In}{Tile Number} == (${In}{Tile Count} - 1)) && ${In}{Floor Number} > 0

Value ← Floor Number

Figure 4.1: Building model featuring balconies stretching across facade corners, an example that cannot be naturally achievable using shape grammars.

In the shape domain, one must additionally distinguish the concept of *merging* and *unification*. Merging is the process of collecting all faces, edges and points of different shapes into a single shape entity, while unification does the additional step of finding and connecting common face vertices and edges.

A useful application of these operations is portrayed in Figure 4.1, featuring balconies stretching across corners - a recurring issue hardly achievable using shape grammars. The facades are split into a grid of separate tiles, each containing information about their X-Y index within the grid of the respective facade. These are used to conditionally filter the first and last tile from each facade, for non-ground floors. The 'Group' node then assembles all these tiles by floor. Within each group, tiles are merged and unified if they have overlapping edges (using the 'Adjacency Merge' node). Having the corner faces together within the same shape, extrusions along the average of the faces' normals create the seamless result intended for such balconies.

Spatial clustering is another form of grouping that is illustrated in Figure 4.2. Given the lots of the peripheral area of the generated city, six random lots were selected as initial centroids and, using a k-means algorithm, the remaining lots were sorted according to its proximity to the cluster centroids. The obtained clusters were then used to construct alternating industrial and residential neighborhoods, with green areas located at the cluster centers.

4.2.2 Aggregation

One of the possibilities that emerge from controlling over sets of objects is the application of aggregation functions - minimum, maximum, average, sum - over entity properties or attributes. They are frequently used with condition or grouping nodes to filter and organize entity sets.

Figure 4.2 displays a complex example featuring an extensive urban environment with thousands of buildings. The starting point was a list of street blocks, each defined as a shape entity. For each one, the geometric centroid was calculated and stored as an attribute. Using the 'Aggregator' node, the whole set of blocks was collected, the centroid attribute taken as a value to average and the result stored as the 'City Centroid' - the city center. The distance calculation to the center is performed for each block individually but, in order to calculate the relative distance (a value between 0.0 and 1.0), the maximum value was determined using again an aggregation node. City zoning was then determined by this relative measure: ≤ 0.3 for the downtown area, ≤ 0.45 for the commercial area and the rest for the peripheral area. The height and style of buildings is random within the range acceptable for the assigned zone.

After the aforementioned clustering operation for the peripheral area was applied, the center of the centroid was calculated using the same aggregation method.

4.2.3 Amount Filtering and Counting

A very important flow control feature, unaddressed by shape grammars, is the ability to determine the amount of entities flowing at a particular point of the graph. This measure is especially important for filtering entities according to absolute or relative quantities. It is also required

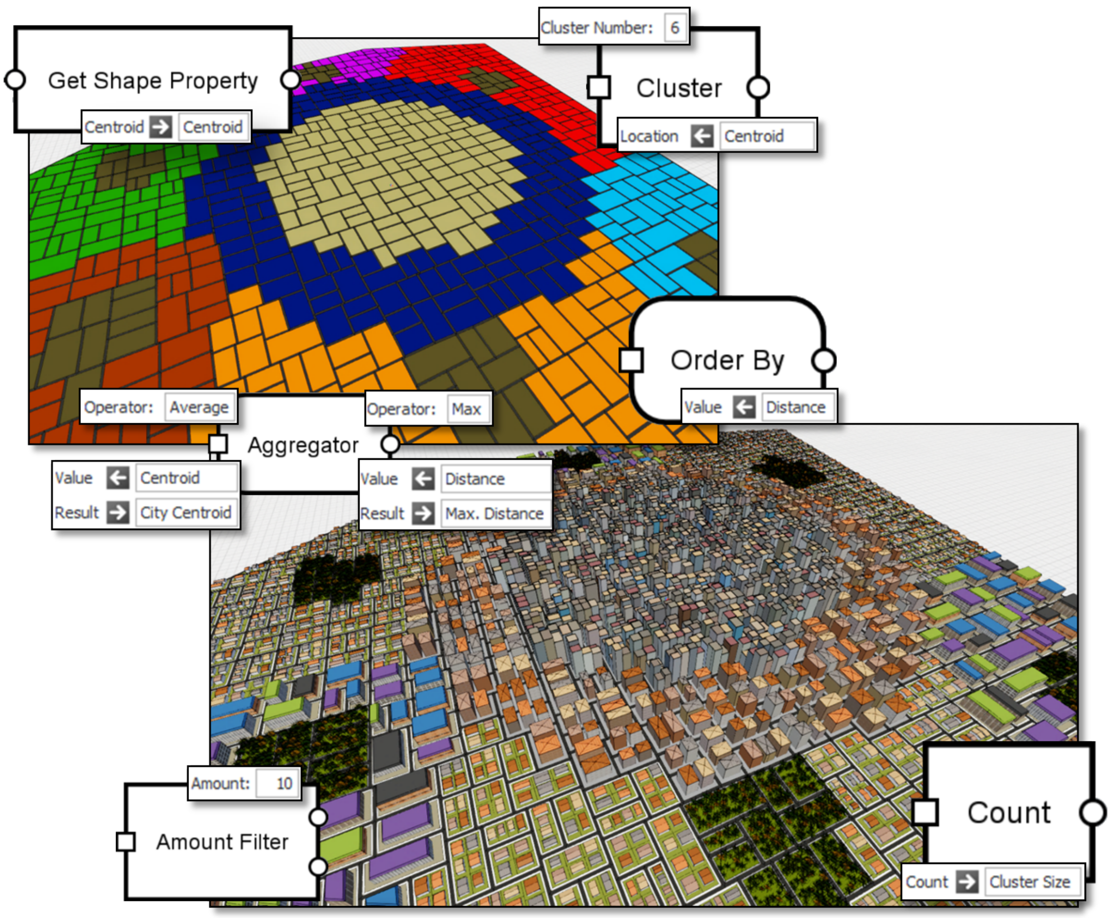


Figure 4.2: Organization of city blocks for construction. After randomly generating about a thousand city blocks, the city center was determined in order to build a downtown area, a surrounding commercial area and a peripheral area. The latter was divided into six clusters to define alternating industrial and residential neighborhoods. Again, the center point of each cluster was then calculated and the ten closest blocks were reserved for green areas. This process involved several aggregation, sorting, clustering, grouping and filtering operations.

when a sequential numbering or alphabetization of entities is to be achieved.

For the example in Figure 4.2, exactly 10 blocks per cluster were picked for building green areas. The ’Amount Filter’ node introduces this possibility, as it collects entities as a list and isolates the first n entities, starting at a given index i, and forwards them to its first output port, while the remaining ones are sent to the second port.

The need to isolate one instance from a set happens often when an particular detail is to be introduced at a given point, e.g. placing the main door at one of many candidate facades of a house. This is demonstrated in Figure 4.3, where a common rule has to be found to address the various possible building shapes. The main door should face the street, yet many options exist (red, yellow and blue arrows). Deciding upon the smallest facades (yellow and blue arrows) reduces the number of possibilities, but to ensure that only one door is created, the ’Amount Filter’ node is required.

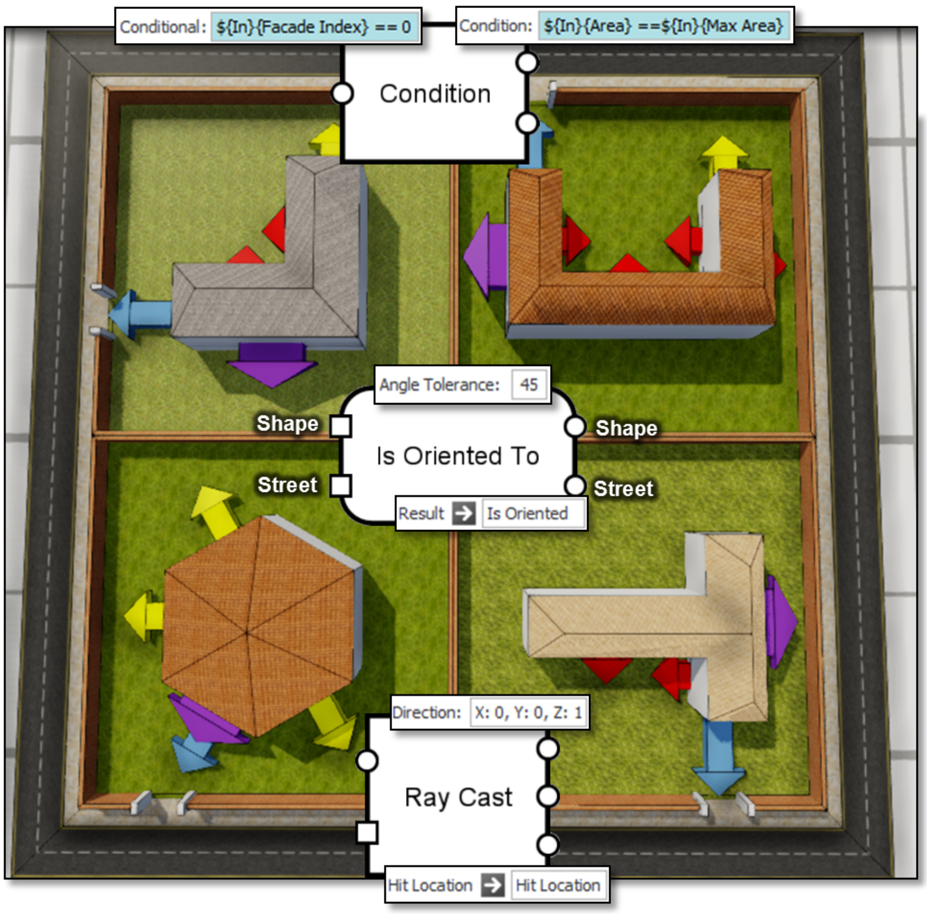


Figure 4.3: Selection of the main door facade for several building footprint shapes. Purple arrows indicate a selection by facade index, very common in systems such as [Sid15], but which do not always guarantee plausible door locations. All other arrows refer to facades that are street oriented (as in [Esr13]). From these options, the smallest facades were chosen using aggregation (yellow and blue arrows), but, to guarantee uniqueness, amount filtering was employed (blue arrows). The selected facade was fed to the 'RayCast' node, which returned the front-facing wall and pointed to the right entity and location for the lot entrance.

4.2.4 Ordering, Reversing and Shuffling

The order in which entities flow throughout the graph may be determinant to achieve a specific result. The 'Cluster' procedure (see Figure 4.2), for instance, automatically picks the first n entities as source for the initial cluster centroids. As such, shuffling the city blocks beforehand ensures some randomness in the clustering process.

The 'Order By' augmentation node features the means to sort entities by several given attributes. When used in conjunction with the 'Amount Filter', it can be used to filter the n smallest facades (as in Figure 4.3) or the n closest blocks to the cluster center (used for green area determination in Fig. 4.2). Likewise, obtaining the largest facades or most distance objects could be obtained through the 'Reverse' node that inverts the entity list order.

4.2.5 Context Awareness

An important consideration is that the generation process should not only depend on the properties of each entity, but on its context as well. In PCGRs, this is best addressed using several input ports, each accepting data according to a specific meaning. Figure 4.4 portrays an environment built around this concept, where the level of detail of each building is determined by its distance to the main path (using a similar approach as Section 4.2.2).

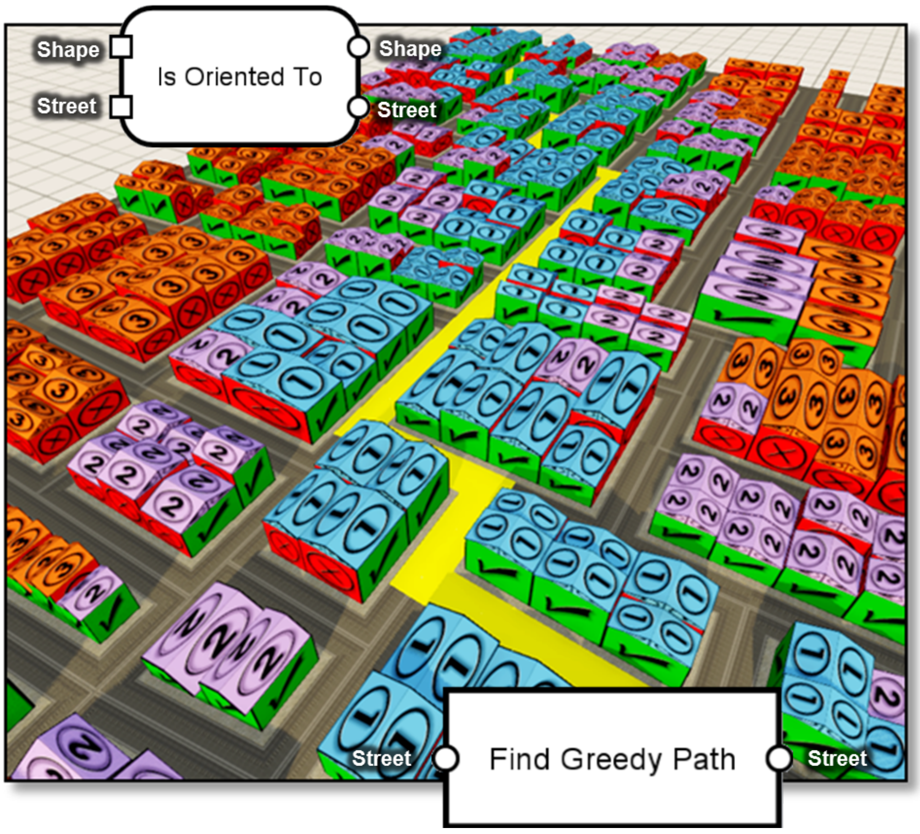


Figure 4.4: This approach introduces context-awareness verifications which can be applied at any step of the generation process. Here, visibility calculations are performed to determine which facades are visible (marked in green) from a highlighted street (marked in yellow and black), and which are not (marked in red). The minimum distance of each house towards that same street is calculated, and the decision on the level of detail (1, 2 or 3, as marked on the roofs) is based on the distance. For applications focused on a main path (such as racing games), this information could be used to guide the procedural content generation and include budget considerations in the level construction.

On the other hand, the decision on whether to design each facade depends on its visibility from the same path, as facades that will not be seen are best left out to reduce the memory and rendering overhead. The operation facilitating this verification, also employed in Figure 4.3, hereby simply named ’Is Oriented To’, accepts a list of shapes and a list of streets and calculates, for a given angle tolerance, if the shape is visible from/to any of the streets, without being occluded by another shape entity. The result is assigned to an attribute, but the distinction could also be done using several outputs. One important consideration is that, by collecting all the shapes and streets at once, this node can organize and optimize the verification internally using spatial data structures, such as quad- and octrees.

Another example node (depicted in the same Figure 4.3) that interrelates entities is the Raycast node. For a given shape arriving at the first port, it casts a ray, following on the scope direction, and searches for the first intersection with the shapes arriving at the second port. The original object is returned from the first output port, while the separation of hit and non-hit shapes is performed to the second and third output ports. The actual hit location is stored to an attribute, which can later be used as a reference point for splitting or other operations.

4.2.6 Combinations

In the process of manipulating sets of entities, it is common to meet the need to pair entities in order to perform direct comparisons, verifications or connections between multiple objects. For instance, one may need to compare all entities in a certain set with all elements of another set. Or perform a verification between all combinations of elements of a given set. This is generally cumbersome task to perform in most programming languages, requiring several stacked loop control cycles that invoke the intended function with each pair of elements.

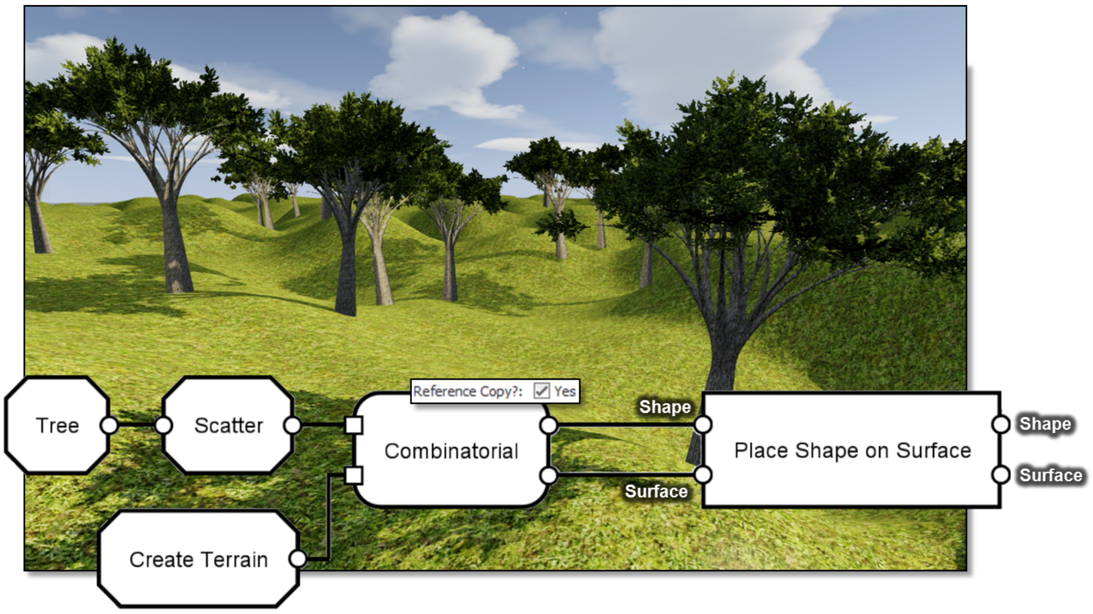


Figure 4.5: Placing several trees on a terrain.

Procedural Content Graphs, inheriting a rather functional paradigm, suffers from the same difficulty, but compensates with the possibility to control entity references. Figure 4.5 illustrates an example where several tree shapes are meant to be placed on one surface terrain. This can be attained through the "Place Shape on Surface", a single node which accepts two inputs, one for the shape and another for the terrain, and places the first on top of the latter. Because of the process of entity processing, feeding a list of elements to the first port and a single element to the second port will simply perform the operation for the first shape and first surface, returning modified versions on the output port, but leaving the remaining shapes waiting at the first input port. To perform the operation for all shapes, a recursive loop has to be created to feed back the modified surface to the second input port again. The number of loops should not exceed the number of shapes, otherwise the surface may be the one waiting at the second input port.

An alternative strategy consists in using a combinatorial node to create all combinations of input entities, outputting reference copies. For the example in Figure 4.5, for each shape, a reference copy of the surface is created, so that the "Place Shape on Surface" node can operate with an even number of input entities. Because the referenced surface entity is the same, the same entity will be consequently transformed in the "Place Shape on Surface" node. There is no need to send the entity back to the input port, hence simplifying the graph design and achieving the same result.

4.2.7 Enforcing Execution Scopes

As mentioned in Section3.1.1, collective nodes attempt to gather as many entities as possible before executing its operation. This may, in some situations, become an issue if one would rather have it executed for subsets of the collected entity list. In this sense, encapsulation does not only come as a means to organize graph procedures, but also to provide an improved control over the data flow.

Figure 4.6 reflects a situation where several houses are generated using a simple graph. Supposing one example where one would try to select exactly one facade of each building to create the main door. If all the the facades are collected without differentiation, and queued at the amount filter node, only one facade of the whole building set would be selected, instead of one per building, as desired.

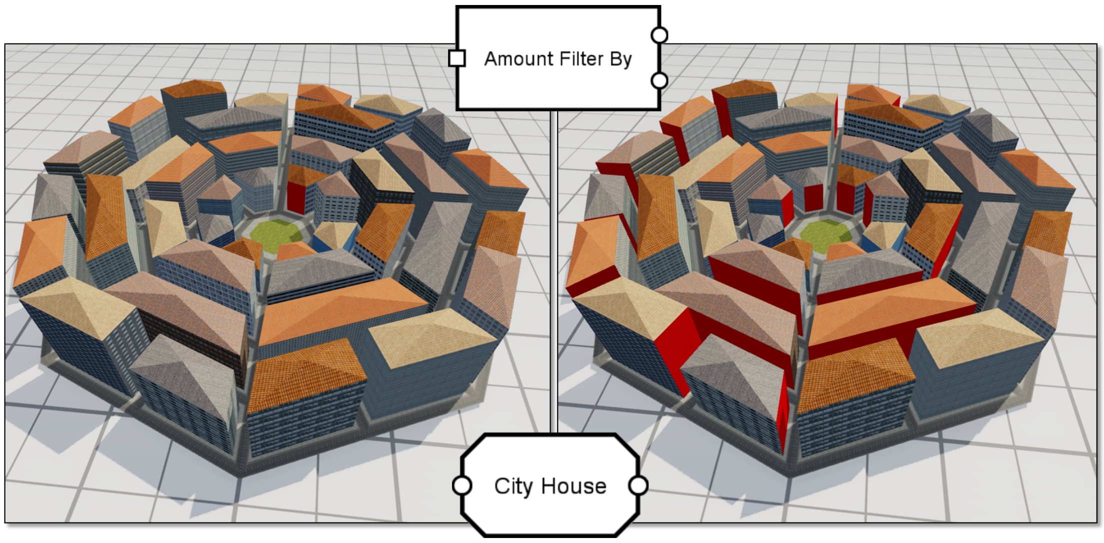


Figure 4.6: Example of the need to enforce execution scopes. If all the the facades are collected without differentiation, only one facade of the whole building set would be selected, instead of one per building.

This can be solved using a "Amount Filter By", which performs that separation internally using attributes, enumerating via node augmentations as grouping criteria (see Section 3.6). This approach can quickly become impractical, if such grouping would have to be performed e.g. per neighborhood, then per building, then per floor. On the other hand, this requires all collective node operations to support such criteria analysis, as well as to have differentiating attributes configured every time.

Another alternative consists in isolating data using encapsulation so as to enforce *execution scopes*. The idea is that the operation over each lot is handled independently. For each input lot, the sequence of graphs, including collective nodes, is performed, meaning that the entity collection of the merge and amount filter operations are executed only within the scope of each individual building. This is guaranteed by the fact that encapsulated nodes, as any other procedures, work isolatedly on its executed queues, before allowing the supergraph to proceed with its execution.

4.2.8 Content Pool

The *Content Pool* is a graph-inherent structure that introduces an alternative data management solution by allowing easy storage and loading of entity data. This is performed through specific store and load nodes: storing keeps a reference to the entity data inside the graph (hence the need for state persistence), associated to a key. Load nodes that are executed at a later moment of that graph or subgraphs can access these entities using matching keys.

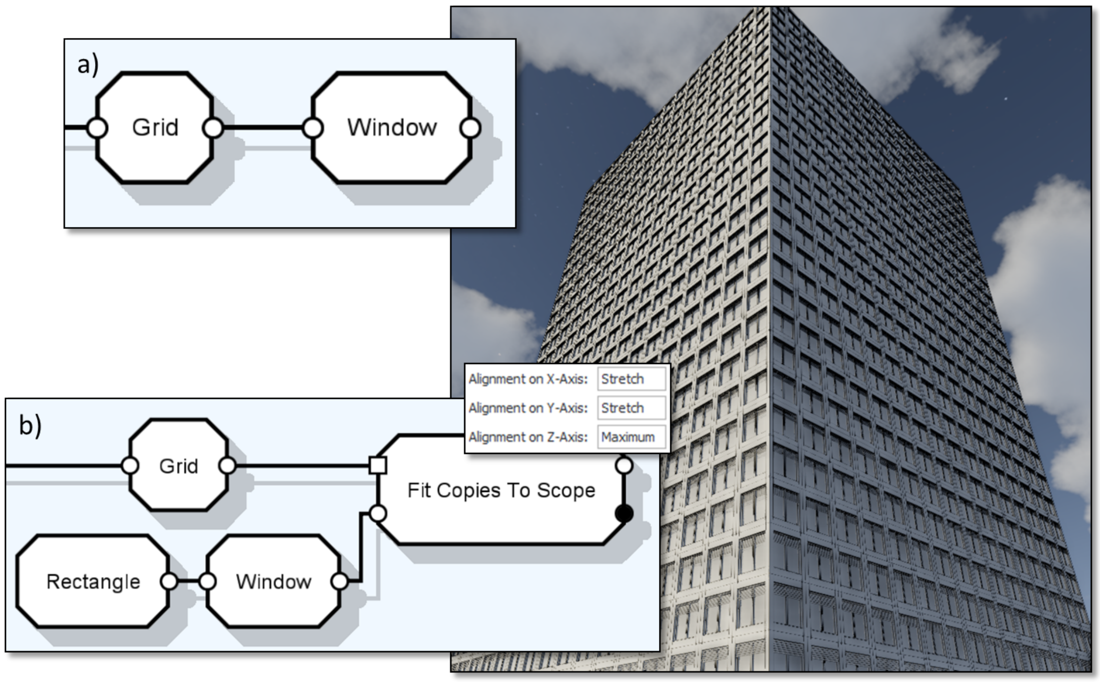
Content pool data is discarded once the graph that holds that information finishes its execution. Both loading and storing can be performed in different scopes, i.e. it is possible for entity data to be stored/loaded to/from the supergraph or subgraph directly. This introduces a means to define global entity data, a practice with all its benefits and downsides.

This pool is nothing more that a convenience which avoids having direct edge connections that could otherwise become too cluttered and confusing to understand in a more complex graph. It also spares additional port definition in the encapsulated graph, as data can be stored to the pool in the supergraph and loaded within the subgraphs. As a result, it can also be used as an alternative to optional ports (see Section 3.3.2).

4.2.9 Avoiding Process Redundancy

The most common strategy in shape grammars is to progressively transform simple entities into more complex ones, such as to transform a simple polygon into a complex window with ornaments via a sequence of splitting, offsetting, extrusion and texturing operations, among others.

Another process that is often used consists in simply loading an external resource/model and then adjusting it to the scope of the parent shape. This has the advantage of being faster to process, as there are less procedures being executed. On the other hand, it is less powerful, as the external resource cannot be parameterized in the way that procedural processes can. A solution is to have a procedural process generate that resource once and load it later multiple times without requiring the execution of heavy geometry generation steps afterwards.



Alignment on X-Axis: Stretch
Alignment on Y-Axis: Stretch
Alignment on Z-Axis: Maximum

Figure 4.7: Alternative methods to apply the same window design on a grid-based building. The solution b) requires only one execution of the "Window" node to produce one copy, which is then replicated to the remaining grid positions.

This is the case of the window in Figure 4.7, where the facades have been cut into equally sized tiles, meaning that the operations that are applied over them may be simple variations or transformations of each other, having different scales, orientations or translations. So, instead of repeating the same process, the window generation node can be executed only once, while its produced model will just need to be copied and adjusted to the corresponding location. This is possible though the "Fit To Scope" node, which features two ports with one accepting a list of objects with defined scopes (which indicate position, rotation and scale) and will copy the shape arriving from the second input port into those scopes. This is a case where process redundancy can be avoided, achieving the same result with much greater efficiency.

4.2.10 Constraint-Based Generation

Constraint-based procedural content generation is a popular alternative to pure imperative-based approaches. The main idea is that one can provide a set of constraints that define what content should be produced, instead of how. This is a rather declarative approach, where it is not necessary to indicate the sequence of generation operations. In many cases, this is the best option, as many generation techniques require a trial and error approach, featuring backtracking, as well as other strategies that would be very difficult to implement using grammar or dataflow-based approaches.

Still, constraint-based approaches have other downsides. In many cases, the design of many structures using constraints is more cumbersome, as many may have to be introduced to achieve a specific detail, which would otherwise be more rapidly defined using imperative approaches. There are ideas and instructions that are hard to express semantically and are more easily described using sequences of modeling operations. Such is the case of building ornaments, for which sometimes no describing vocabulary is known or even available.

Another downside is the time to evaluate the constraints and produce a valid result. For large and complex cases, constraint-based programming can result in considerable overheads for processing, making the design of full-scale virtual urban environments unfeasible.

One of the domains where constraint-based programming has proven to be particularly effective in the production of virtual urban environments is in the automatic design of building interiors [Lop+10]. Often, a designer simply knows the nature of the intended rooms and the allowed connectivity between them. Instead of going through the effort of building such layouts himself, these indications could be fed to an automatic layout solver that would quickly find the best solution. Another example of good application is in the automatic placement of objects within rooms [Tut+11]. One simply needs to indicate what kind of objects are to be placed and where, and the layout solver can find several alternatives.

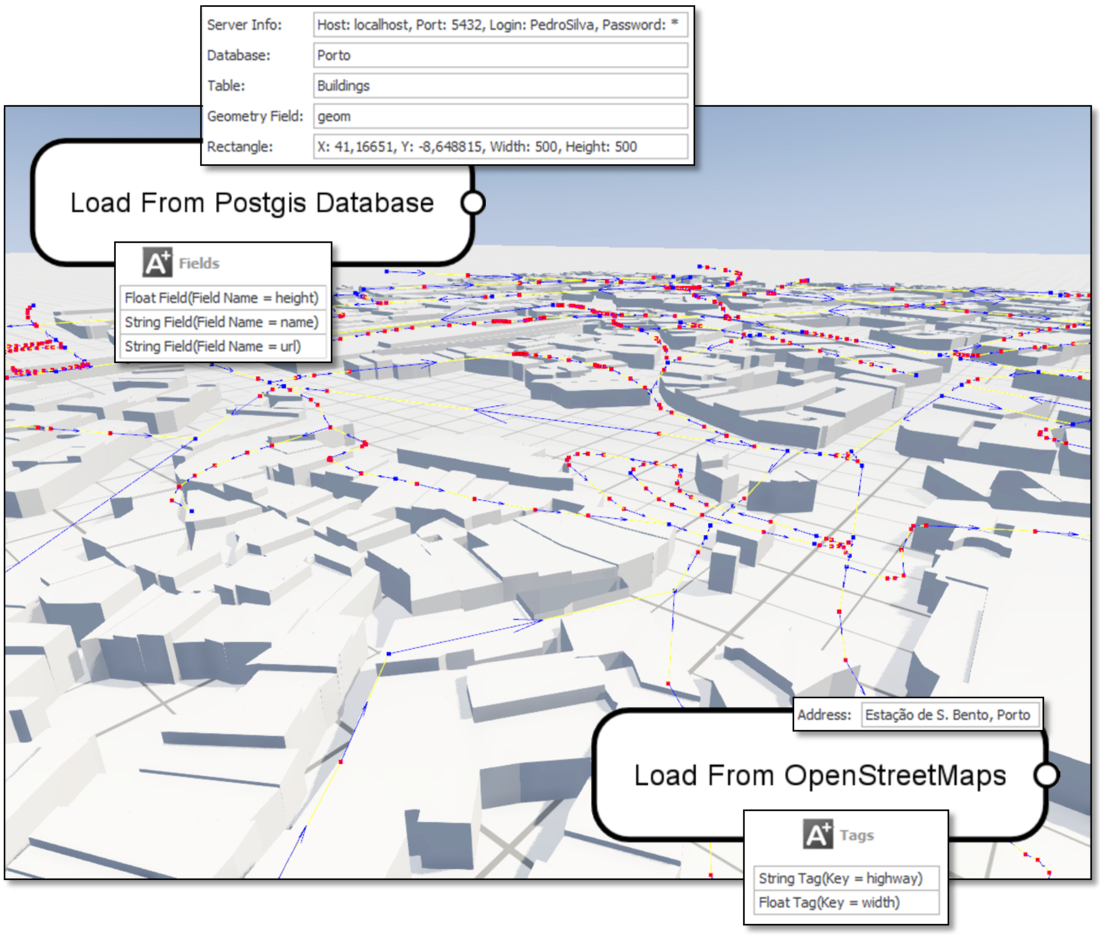
In order to avail the best of both worlds, PCGRs can very easily integrate constraint-based approaches with standard imperative-based approaches, where the user must indicate the steps to perform. The use of augmentations is practical, since they allow the enumerations of the said constraints, as well as their parameterization. Constraint-based operations become therefore restricted to particular nodes, and can easily fit with the remaining methodology.

4.2.11 Integrating External Sources

The generation of virtual urban environments can be guided by real-world data sources, such as photographs, maps or any other type of parametric information, in order to produce results that have a high level of correspondence to real world locations. A problem, however, lies on the integration of several data sources, since each one can incorporate different formats, structures or its own descriptions. Besides, this data sources are constantly subjected to inconsistencies which derive from their complexity and dimension. Although their capture is mostly automated, there is a considerable amount of manual post-processing labor involved, all of which affect the quality of the data.

This problem and difficulty has been contemplated by the work of [Coe+07], which encompassed a set of XML definitions to describe the sources of data and how they could be handled and processed, but the development of such definitions proved to be tiresome and complex to write. The work of [Mar+12] went further so as to try to map information to an urban ontology, defining levels of mapping to categorize the amount of information available to describe a building. Even so, the chaotic nature of many data sources has made this approach difficult to employ in many situations.

A solution, fitting with Procedural Content Graphs, consists in the conception of several nodes, each addressing a type or format of data source, but converting them to known entity type, manageable within the graphs. Although certain generic strategies can be employed, the fact is that specific loading and transformation processes are often required. For the graph designer, this requires simply the choice of node (typically, a source node), which will then be chained to the remaining process. As long as the produced entities are the same, changing the starting point does not require a alteration in the remaining graph.



Server Info:	Host: localhost, Port: 5432, Login: PedroSilva, Password: *
Database:	Porto
Table:	Buildings
Geometry Field:	geom
Rectangle:	X: 41,16651, Y: -8,648815, Width: 500, Height: 500

Float Field(Field Name = height)
String Field(Field Name = name)
String Field(Field Name = url)

Address:	Estação de S. Bento, Porto

String Tag(Key = highway)
Float Tag(Key = width)

Figure 4.8: Loading GIS data particular types of nodes that address the specifics of the sources.

Figure 4.8 shows an example of a generic node that accesses OpenStreetMaps, an online source of geographic information, and another that loads data from a PostGIS database. The data, returned in the form of either points, lines or polygons, can have associated parametric data to each geometry. These loading nodes are augmentation nodes, whereas their augmentations define what fields should be loaded and to which attributes they should be mapped. As a result, the geometries are converted to Shape entities with associated attributes, meaning that they can be used to serve the later steps of the generation process of the urban environment.

Evidently, this could lead to a true explosion of data loading nodes, each dedicated to a different type of data. However, even such an issue can be minimized though encapsulation, by assembling the many loading nodes into options of a complex one. Alternatively, the use of dynamic loading nodes can equally simplify the process, congregating the many data loading nodes into a single node that chooses the proper process according to the type and format of data, returning the corresponding content entities.

4.2.12 Manipulating and Extrapolating LODs

In 3D geometry, producing progressive levels of detail for the same model is a common strategy for rendering optimization. Models that are rendered at greater distances do not require a great geometric detail, as they cannot be perceived anyway. So the idea lies on providing several versions of the same model, but featuring progressive levels of geometric detail between which the rendering engine can then alternate to improve rendering performance. However, creating all this versions is generally a tiresome task in manual modeling. There are many algorithms that perform mesh simplification, but often some manual labor is still required.

Using grammar and dataflow-based approaches, this can be simplified by just using conditional nodes that redirect the flow based on a global parameter or attribute that control the overall generation process. That value indicates what level of detail is intended, and the whole process simply adjusts by executing more or less operations, producing consequently more or less complex versions of the same model. For example, the low detailed version of a building can be extracted by simply applying a texture on an extruded block. However, producing a texture that is equivalent to the most detailed version can still remain a tiresome process.

In PCGRs, the same mesh simplification algorithms, found in many modeling applications, can easily be chained in the generation pipeline, after the high detailed version is finished. Using merging and unification nodes, the whole model can be joined and simply fed to the algorithms. Albeit the strategy explanation is here simplified, what was before the major obstacle of data congregation is addressed by PCGRs.

Another idea, first explored in [Ali+09], is the automatic extraction of simple levels of details from building facades. In PCGRs, the process consists in requesting an orthographic render of the detailed building facade, and mapping the rendered image as texture of the facade (see Figure 4.9). The "Create Shape Projection" node accepts a shape and, based on the size of the entity scope (see Section 5.1.1), automatically calculates the position and sizes of the camera, before issuing the texture render. Using this node once per building facade can issue repeated textures, meaning that its application should be restricted to avoid process redundancy (see Section 4.2.9).

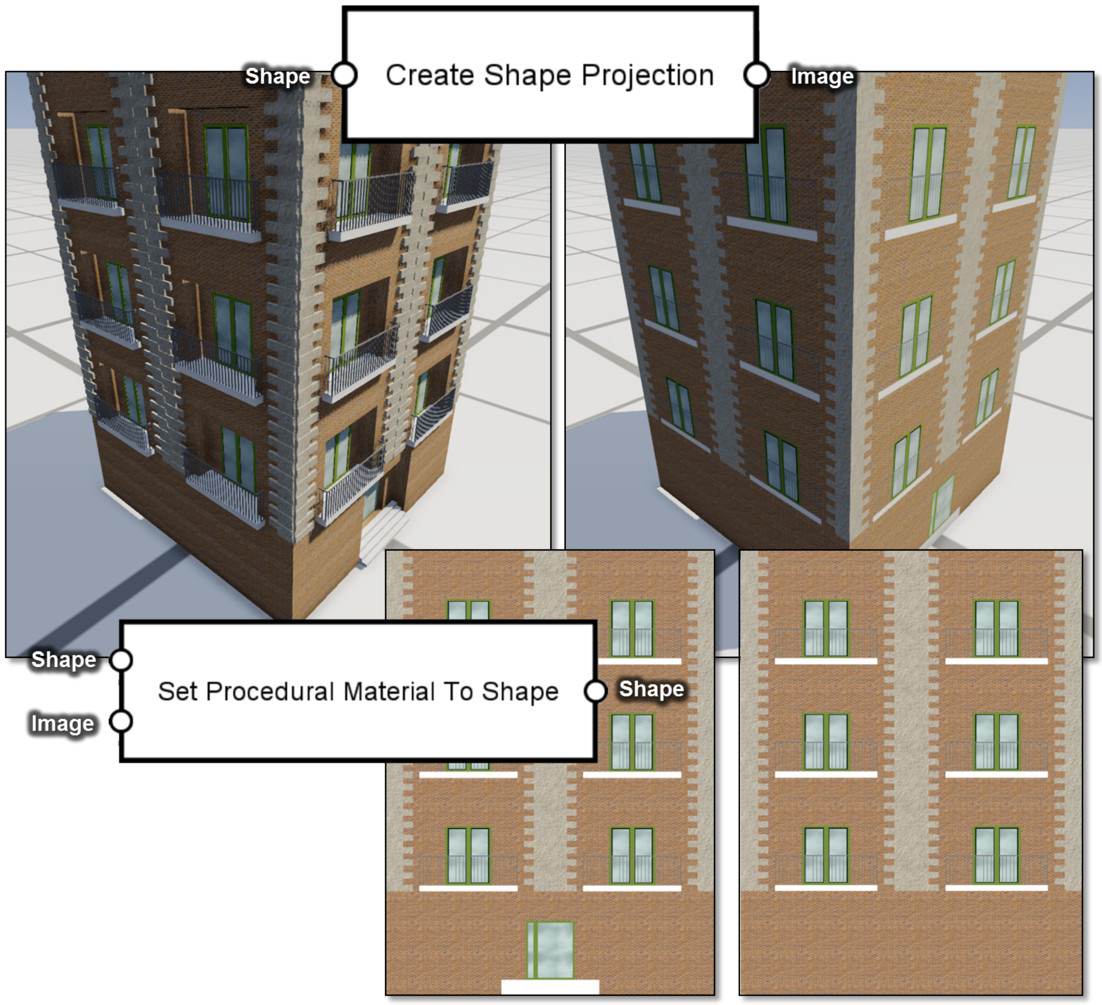


Figure 4.9: Example of the extrapolation of a low resolution version of modeled facades. Using the "Create Shape Projection" node, each facade was rendered to two different textures (shown in the Figure) and applied to a simple geometric block, creating a low-resolution version of the same building.

4.3 Integrating Different Types

The ability to manipulate several types of entities lies on the existence of nodes capable of creating, analyzing or transforming them. This is reflected on the typing of node ports, which provides the means to understand the node's purpose and compatibility. Different entities can co-exist within the same graph and, in some cases, even share the same nodes, ports or edges, if they share the same ancestor type.

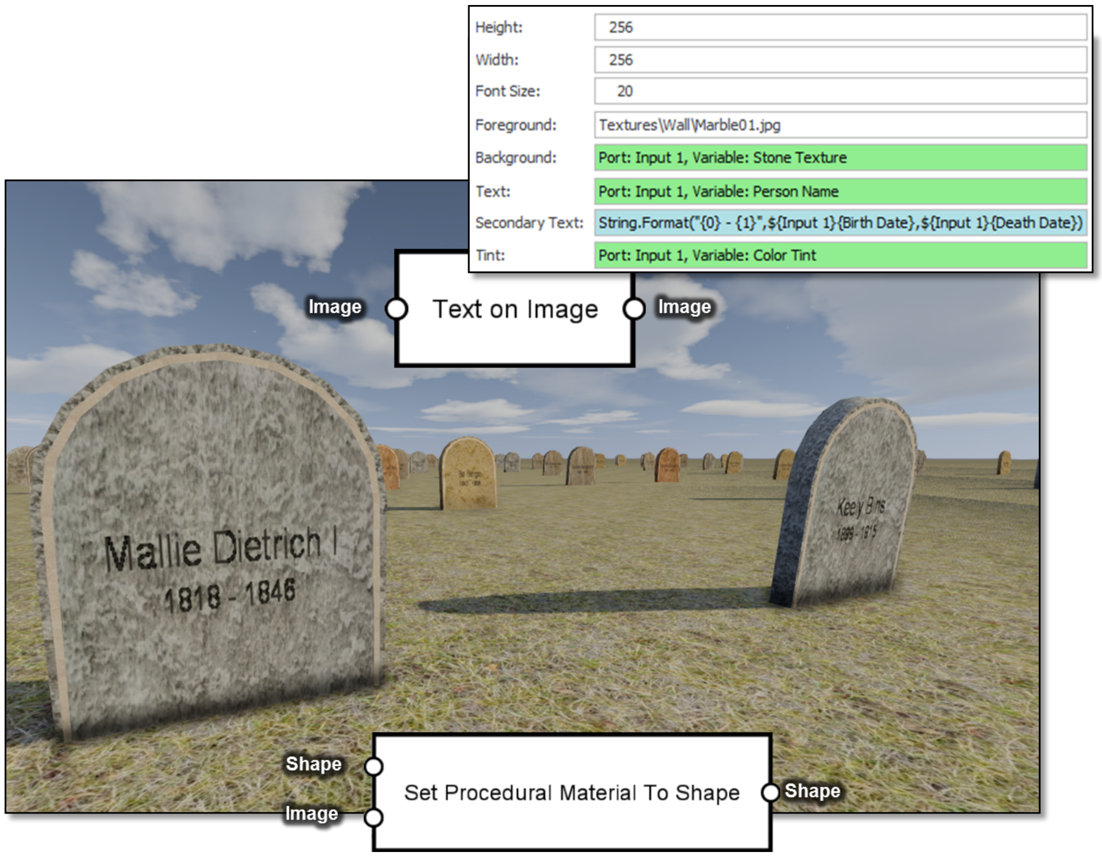
Producing different types of content within the same procedural environment has several advantages, the greatest one being the fact that they can be assembled in the same process, producing a more comprehensive result. When considering urban environments, whose size tends to assume large proportions, developing and putting geometry, textures, lights and behaviors together on a separate step can become a tiresome task, as well as lead to gaps and inconsistencies.

This section will therefore focus on explaining how this integration can be processed and what advantages it can bestow.

4.3.1 Generating Textures

The generation of textures of one example can greatly benefit from an close and chained development with the geometry generation. Textures are a basic ingredient for material rendering, making geometry more diverse, interesting and realistic.

Although, in most cases, the selection of a texture is "generic", being pre-made and applicable to many objects, often they need to be individually adjusted to the object they are being used in. Such is the case for including a custom number, message, label or sign on an object at a certain location or situation, such as a street number, name or other textual information. Figure 4.10 applies this very same concept on the generation of individual tombstones, where the name, date of birth and date of death are randomly generated and laid on a selected texture, using a 'Text on Image' node. Afterwards, this texture is fed to 'Set Procedural Material To Shape' node, which combines texture and shape as a custom mesh material.



Height:	256
Width:	256
Font Size:	20
Foreground:	Textures\Wall\Marble01.jpg
Background:	Port: Input 1, Variable: Stone Texture
Text:	Port: Input 1, Variable: Person Name
Secondary Text:	String.Format("{0} - {1}",${Input 1}{Birth Date},${Input 1}{Death Date})
Tint:	Port: Input 1, Variable: Color Tint

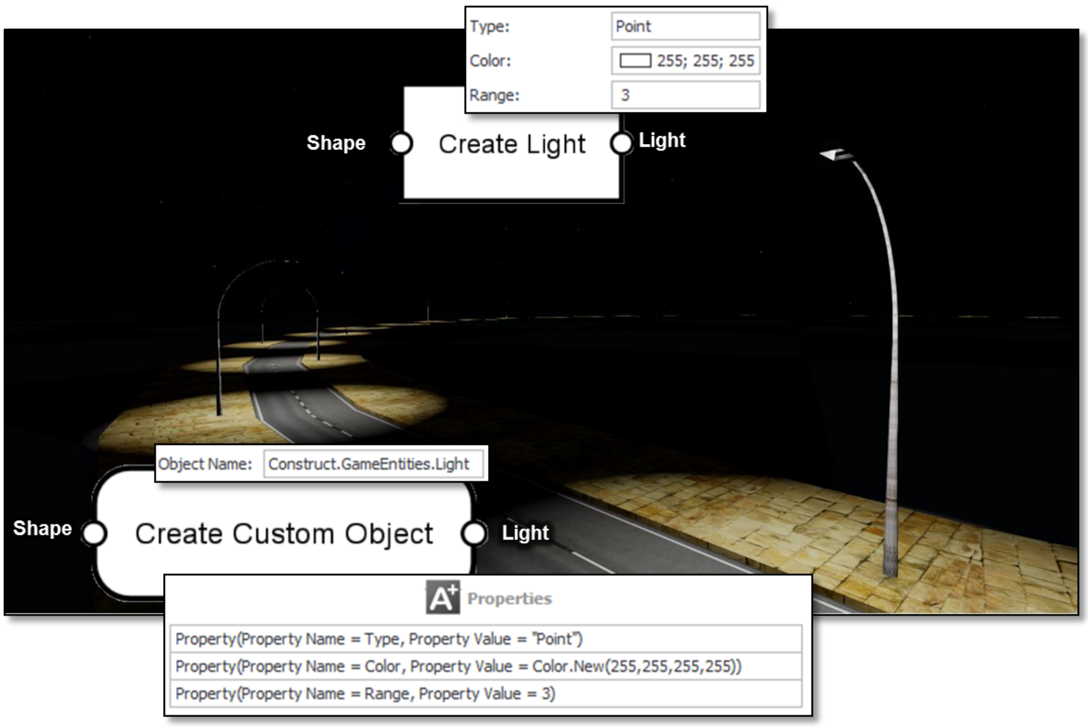
Figure 4.10: Procedural generation of tombstones, each featuring random, yet plausible names and dates.

Another issue associated to generic textures is their tendency for repetition, which can make environments uninteresting and artificial. Here, the introduction of some custom noise or color tints to a pre-made texture can help providing different looks for the same texture. Again, this is represented in Figure 4.10, where the same stone texture is used, but featuring random color variations.

Another example has already been enunciated before in section 4.2.12 as a means to extract a low-detail version of a complex model. Obtaining such rendering could otherwise become a tiresome task if it would have to be developed for thousands of buildings that constitute an urban environment.

4.3.2 Producing Property-Based Objects

In the scope of shape grammar methodologies, the common strategy consists often in splitting objects into smaller sections and scopes, which are simpler to operate on. Based on this consideration, the simple task of instantiating any kind of entities in a relative location to the scope of a certain geometry section becomes a simple task to perform in an automated manner. For instance, the placement of a light source on sets of light posts of a street can follow the generation of the lamp geometry (see Figure 4.11). Considering the arrays of lights to place in large-scale urban environments, this would otherwise become a tedious endeavor, if executed outside the generation process as a manual task.



Type: Point
Color: 255; 255; 255
Range: 3

Object Name: Construct.GameEntities.Light

Property(Property Name = Type, Property Value = "Point")
Property(Property Name = Color, Property Value = Color.New(255,255,255,255))
Property(Property Name = Range, Property Value = 3)

Figure 4.11: Generation of light posts and corresponding light sources.

When it comes to the placement of props, lights, characters, sounds, prefabs, etc. the first step consists in the adjustment of the transformation matrix of the object, i.e. the adjustment of the position, rotation and scale of the objects. In addition, there may be other properties that need to be assigned and adapted to the circumscribing environment. For lights, it may be type of light (e.g. omni, spot, sun), its intensity, color or range, which may need to be tuned to guarantee a proper lighting of a given space.

In addition to a node that creates an explicit instance of a light entity, Figure 4.11 illustrates the "Create Custom Object", a generic node which receives, as a string control parameter, the name of the entity to be created. Using augmentations, their property names can be enumerated and their default values defined. This information is stored within a generic entity as well, which is only materialized when used within the target application. In other words, it is up to the subsequent content managers and game engines to process such entity definitions, by instantiate the actual object by the indicated name and setting its properties according to the defined specifications. Such approach eliminates the need to define specific nodes for each kind of application-specific object.

4.3.3 Producing Component-Based Entities

A popular and flexible architectural alternative in virtual reality applications consists in representing entities as containers of *components*, where each holds a specific set of information (e.g. mesh, material, mass) or a particular behavior (e.g. animation, interaction, trigger). This approach has the advantage of making certain functions and characteristics far more reusable and flexible, eliminating the ambiguity problems of deep and wide inheritance hierarchies.

From the point of view of generation, this implies the use of nodes that instantiate, configure and then add such components to an incoming container entity, hereby simply named *application entity*. One possible management approach consists in handling a component as an entity, meaning that it can be instantiated by a particular set of source nodes, flow within the graph and later added to an application entity using dual input port nodes. From a semantic point of view, however, this may not make much sense, as components generally do not correspond to meaningful, self-contained objects, but only when incorporated within the application entity.

An alternative strategy consists in viewing both application entities and components as a stack or queue of configurable elements. A node that changes properties from mesh components will access the last added component of that type. This way, managing the container entity becomes equality powerful, but requires a simpler graph construction, with less nodes and edges (see Figure 4.12).

It is important to note that, in certain cases, components may encompass further enumerations, which need to be customized as well. Such is the case of an *animation* component, where animations may be composed by several transformations, namely translations, rotations and scales (see Figure 4.13). These have to be organized into groups, which indicate the number of repetitions and, more importantly, if the transformations are to be performed simultaneously or in a sequence. Such groups can be stacked recursively to more complex animations. Again, the same two data handling strategies apply.

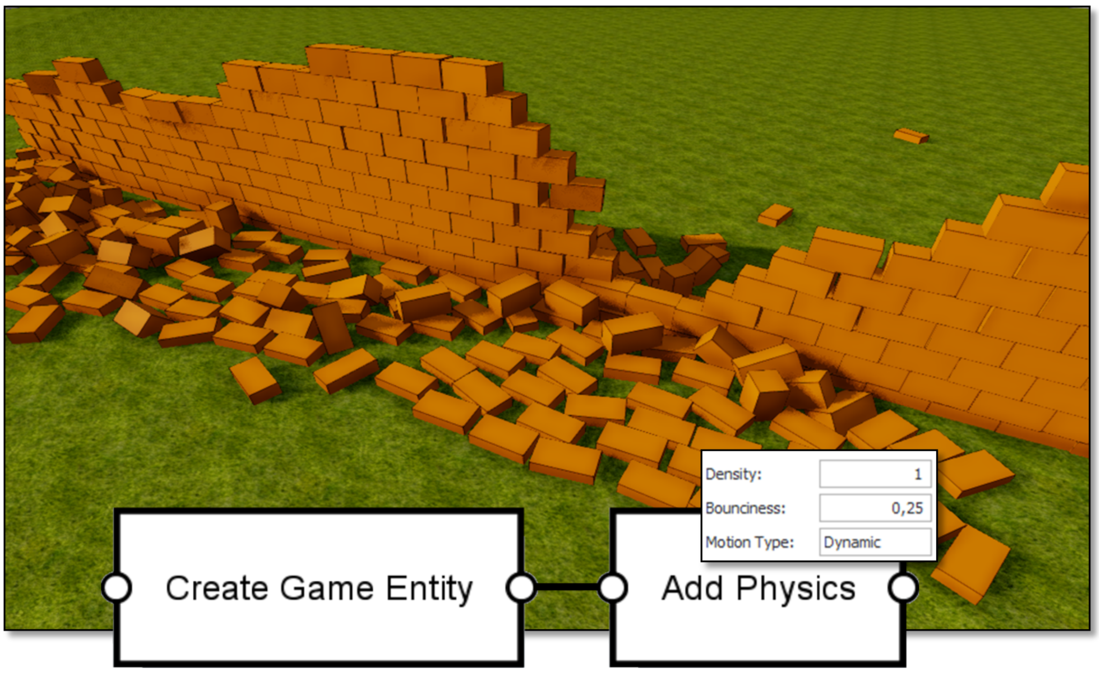


Figure 4.12: Generation of a brick wall, whereas the physics properties are added as a component to each brick object.

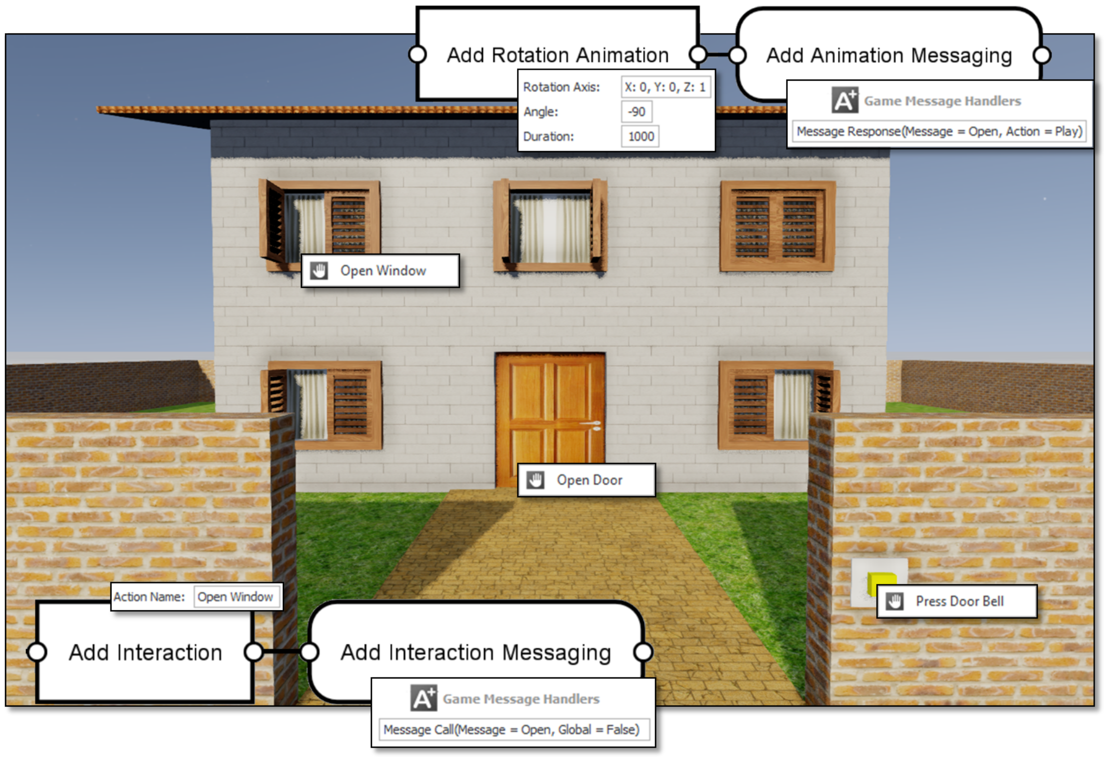
4.3.4 Adding Entity Interaction and Triggering

In many cases, application objects have certain associated behaviors that can affect or be affected by other ones. For instance, a light switch, when triggered, will toggle a light switch on or off. Also, the approximation of a character to a trigger area next to a set of automatic doors will cause the doors to open. Or, the detection of an enemy by a camera will cause alarms to be activated. All these examples require the proper pairing or connection of entities, so that they can properly operate and communicate during the actual gameplay or simulation.

An approach consists in having certain objects hold references to objects with which they have to communicate or trigger. For instance, a keypad that opens a door has to hold the reference of the door it is supposed to open or unlock, once the proper key code is introduced. The device that is capable of deactivating the alarms of a sector must hold the references to those alarms and those only. The number buttons in a elevator must lead it to the corresponding floor, meaning that not only the reference of the elevator, but also the floor location must be stored.

The management of such pairing and communication is achieved using *messages*, which can trigger one or more component *actions*. Such is the case, for instance, for the animation component, whose animations can be played, paused, restarted or stopped. Components that can be triggered are configured via augmentation nodes, where a list of message-action *responder* items can be added and edited. When a message, identified by a particular string, is received by that component, the corresponding action is performed (see Figure 4.13).

On the other hand, there are certain component events, moments or action that can, in turn, set off the broadcast of messages. For instance, an animation can send a message when it



Rotation Axis: X: 0, Y: 0, Z: 1
Angle: -90
Duration: 1000

Message Response(Message = Open, Action = Play)

Action Name: Open Window

Message Call(Message = Open, Global = False)

Figure 4.13: Trigger setup to allow door opening, window opening and door bell ringing. Interaction components, listening to mouse interactions, are configured to send messages instructing the corresponding door to be opened or closed. Animation components are configured to play opening and closing animations whenever such messages are received.

reaches the end or a certain step of the animation. A mouse trigger component will simply send messages when a certain type of mouse interaction is performed upon it (e.g. pressed left button, released right button, scrolled middle wheel). Such component configuration is also achieved via augmentation nodes, where event-message *caller* items are managed.

Proper pairing of trigger and triggered components can be achieved in two ways. On the one hand, by clarifying the recipients of the messages: a message can be broadcast globally to all components of all application entities or locally to the components of the same application entity. On the other hand, by composing more complex messages, featuring unique identifiers to filter the objects that must be addressed.

An illustration of this configuration is displayed in . [Explanation will ensue later].

4.3.5 Producing Challenges

In the scope of certain applications, such as games or training simulations, once the mechanics have been programmed, the longest task consists in adding tasks and challenges to fulfill, which basically means that sets of objects with associated behaviors have to be spread in a consistent and interesting manner.

Depending on the purpose of the game, this requires employing various concepts presented in previous sections. For instance, in a racing game, first the proper racing path must be determined, so that, for instance, appropriate checkpoint locations can be placed in key locations, barriers can be risen and the proper surrounding environment can be assembled.

Many games and applications include, as side quests or goals, the discovery of scattered objects, such as riddles, treasures or activities. Puzzle-based activities are other examples that can enjoy from procedural approaches, where codes and combinations of triggers can be generated to propose an almost infinite number of challenges to the user.

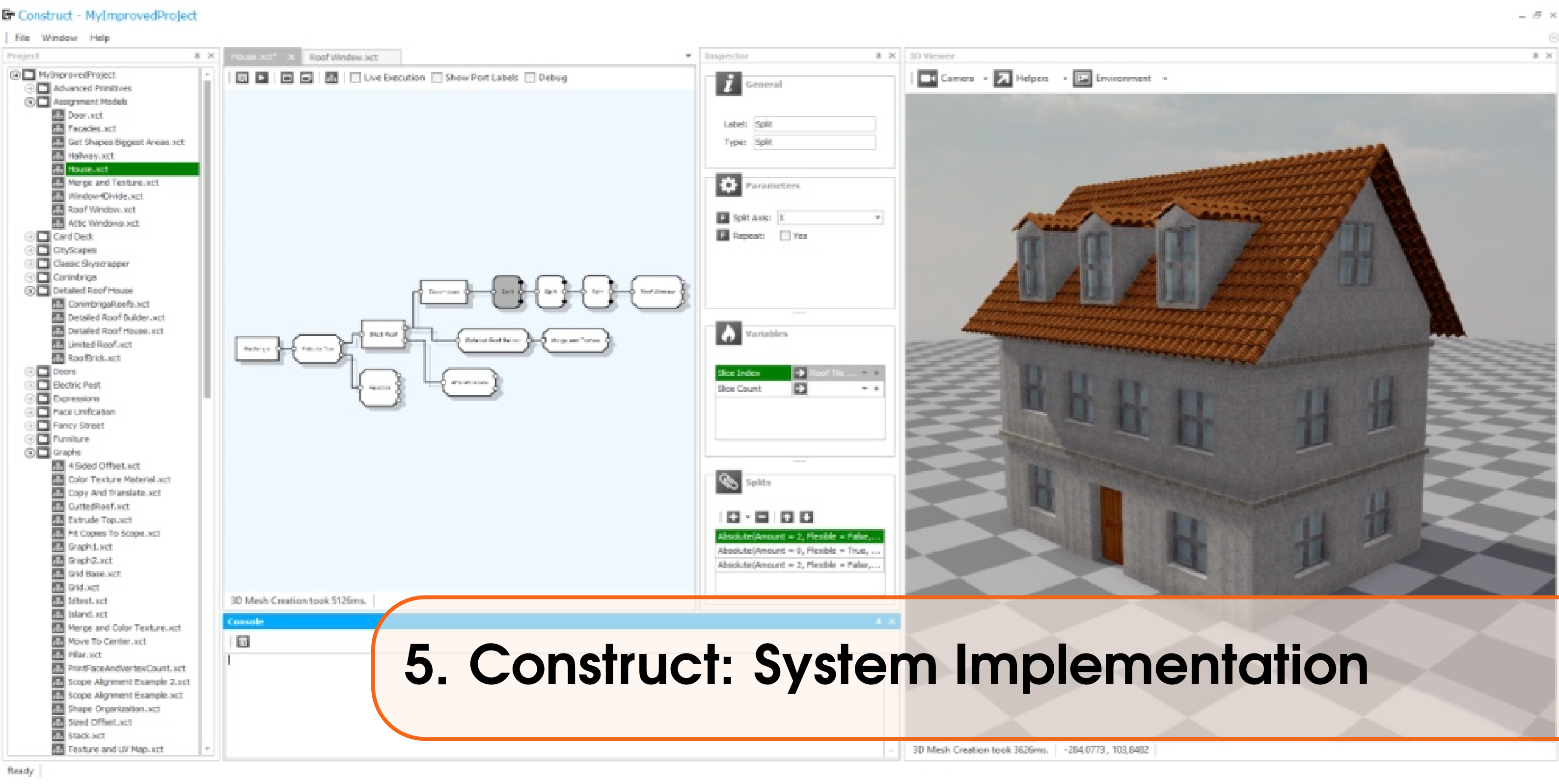
The concept of procedural quests and storytelling involve often the instantiation of Non-Playable characters and quest items in certain circumstances, as well as their interconnection, so that corresponding variables and events are triggered in the right occasions. Evidently, this is no simple task, but can also be automated.

An evident risk of automated challenge generation is the production of repetitive and unimaginative content, which can become dull and unattractive with time. As such, it is important to note that these techniques should not be used to produce the main source of entertainment for such applications, but rather provide more content for the user to remain engaged or even train some specific skills.

4.4 Summary

In this chapter, the possibilities and benefits introduced by the new methodology were explained and illustrated via examples of tasks that were either very cumbersome or simply impossible to express with previous approaches. Such was the case of several operations that required a control over lists of entities, such merging, aggregation, sorting and combination, among others.

Other difficult achievements were also proven much more accessible to achieve, such as the incorporation of external data sources, the extrapolation of levels of detail and the combination with constraint-based approaches. Lastly, the possibilities of data integration were enunciated, providing insight on how different types of content can be assembled in the same process, producing more comprehensive results.



5. Construct: System Implementation

The procedural content graph approach has been materialized into a prototype system called ***Construct***, which consists of a framework and a visual editor. The name had its origins from the first *The Matrix* movie, referring to the virtual reality program that was used for training. In a similar fashion, the system is intended for the generation of the complete virtual environments, from the necessary virtual objects to the challenges and simulation themselves.

This chapter will provide details on the developed architecture, functionality, and possibilities. It will start by explaining how the framework is built, providing some insight on how it can be extended via procedure libraries. Only after will the already developed libraries be enunciated, describing the included entity types and implemented procedures. Later, the visual interface will be presented, explaining the many included features and how they are used to design and manipulate Procedural Content Graphs. The chapter will finalize by discussing how the framework can be connected or even incorporated within other tools.

5.1 Framework

The Construct Framework is a software abstraction that provides the means to procedurally generate virtual content, enforcing the methodology of procedural content graphs. The framework is composed of a set of assemblies developed using .NET C#.

The major **Core** assembly concentrates the definition of procedures, augmentations, entities, aspect types and metadata annotations. It includes a series of basic procedures that operate generically over the entity class objects, hence being applicable to all entity subtypes. It also specifies how the basic data types (e.g. string, integer, double) are to be handled in aspect declarations and what type of constraints (e.g. range, size, format) are compatible with such types.

The Core describes the structure of Procedural Contents Graphs, along with nodes, edges, ports as well as the means to load, save and convert them to executable procedures. Graphs are stored in XML format, making the persistence process more versatile and robust.

Another Core feature is the resource management engine, which allows for proper integration of the framework within different environments. The objective is to provide a generic and transparent method for accessing and modifying external resources, regardless of the platform being used to execute the procedures (e.g. Windows, Linux, Mobile devices). Same applies to the provided message management engine, which allows communication between procedures and framework modules in a decoupled and flexible manner.

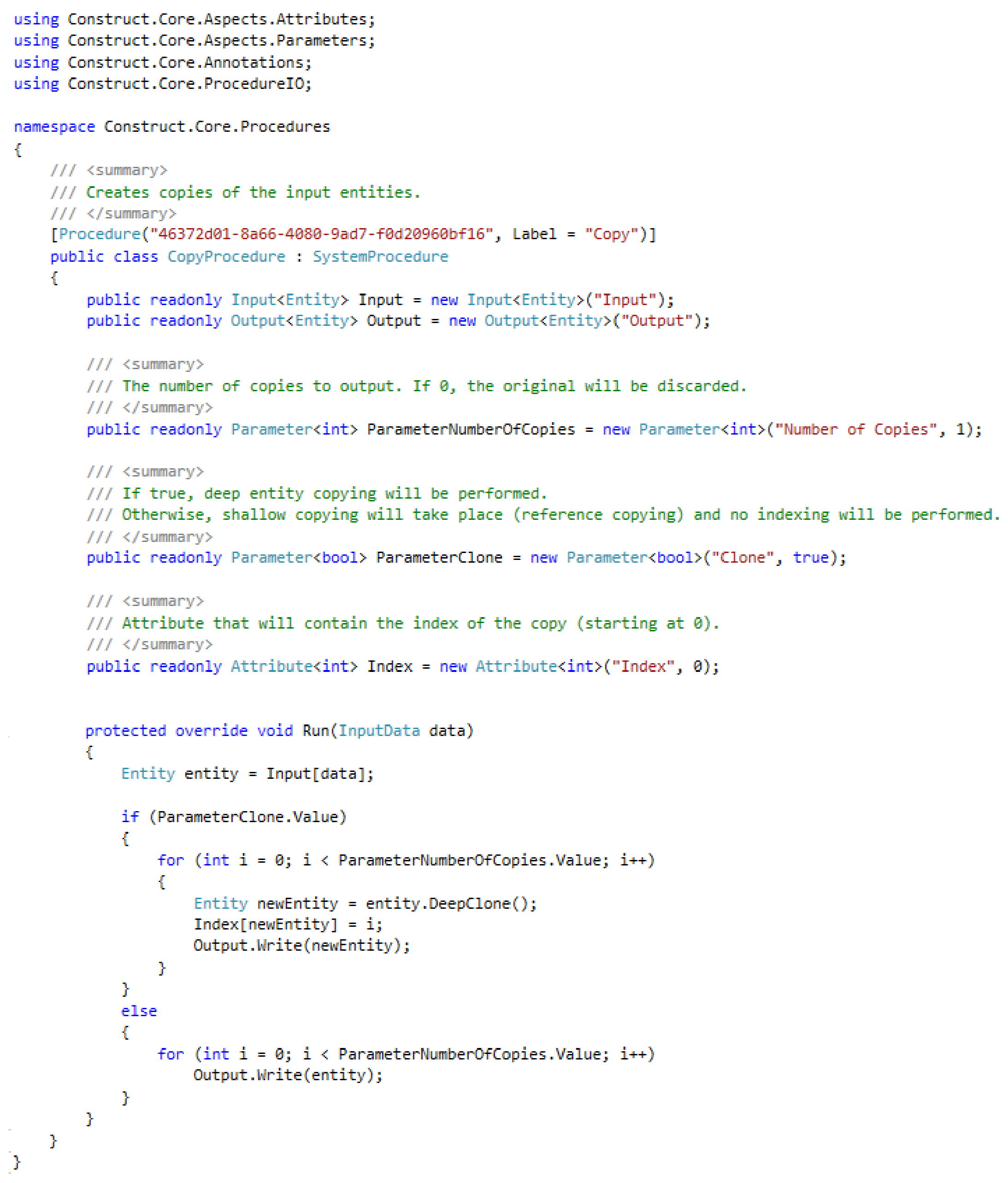
The Core is the basis of the Construct API, which allows the development of new procedures, entities, message and aspect types in the form of *Procedure Libraries*. This extensibility is achieved by overriding certain class declarations, by implementing certain interfaces or by adding certain annotations to classes, members or fields. By using C#'s Reflection abilities, both library data and documentation are automatically read, extracted and initialized, all with the purpose of facilitating the development of the system. Procedure Libraries are meant to be used within any system supporting C# assemblies, ranging from game engines to mobile device applications.

Figure 5.1 shows an example of the necessary code to implement the "Copy" procedure, which receives any kind of entity as input and produces a certain number of shallow or deep copies (whose number is indicated on a parameter), which it sends to the only output port. For the latter case, sequential index numbers are assigned to the clones using an attribute, which can be forwarded to the containing graph.

The addition of a node to a procedure library is confined to such a class declaration, i.e. no further indications are necessary for it to become available to use within graphs. Classes only need to inherit from "SystemProcedure", implement the "Run" function and feature the "Procedure" annotation, where a unique code and a node label must be provided. This code is important to identify the node uniquely, allowing for name and signature changes to be performed without affecting graphs that reference the node.

Inputs and outputs are automatically read from the class fields and converted into node ports. Same occurs for parameters and attribute declarations, which are turned into control aspects with defined default values and constraints. All field and class comments are automatically extracted as node documentation.

At the moment of the current thesis writing, several procedure libraries have been implemented in order to address the generation of many types of content. These shall be presented in the following sections. The intention is to make the framework available for a larger community, making it easy for users with programming experience to add new, lower-level procedures.



```
using Construct.Core.Aspects.Attributes;
using Construct.Core.Aspects.Parameters;
using Construct.Core.Annotations;
using Construct.Core.ProcedureIO;

namespace Construct.Core.Procedures
{
    /// <summary>
    /// Creates copies of the input entities.
    /// </summary>
    [Procedure("46372d01-8a66-4080-9ad7-f0d20960bf16", Label = "Copy")]
    public class CopyProcedure : SystemProcedure
    {
        public readonly Input<Entity> Input = new Input<Entity>("Input");
        public readonly Output<Entity> Output = new Output<Entity>("Output");

        /// <summary>
        /// The number of copies to output. If 0, the original will be discarded.
        /// </summary>
        public readonly Parameter<int> ParameterNumberOfCopies = new Parameter<int>("Number of Copies", 1);

        /// <summary>
        /// If true, deep entity copying will be performed.
        /// Otherwise, shallow copying will take place (reference copying) and no indexing will be performed.
        /// </summary>
        public readonly Parameter<bool> ParameterClone = new Parameter<bool>("Clone", true);

        /// <summary>
        /// Attribute that will contain the index of the copy (starting at 0).
        /// </summary>
        public readonly Attribute<int> Index = new Attribute<int>("Index", 0);


        protected override void Run(InputData data)
        {
            Entity entity = Input[data];

            if (ParameterClone.Value)
            {
                for (int i = 0; i < ParameterNumberOfCopies.Value; i++)
                {
                    Entity newEntity = entity.DeepClone();
                    Index[newEntity] = i;
                    Output.Write(newEntity);
                }
            }
            else
            {
                for (int i = 0; i < ParameterNumberOfCopies.Value; i++)
                    Output.Write(entity);
            }
        }
    }
}
```

Figure 5.1: C# Code for the implementation of a Copy Procedure.

Therefore, it is expectable that the number of available operations will grow further.

The Core library already includes several generic procedures, which can be categorized as following:

- **Organization**: Copying, grouping, and counting entities;
- **Attribute Handling**: Setting attributes, copying attributes between entities, aggregating attributes;

- **Sorting**: Ordering by attribute, shuffling, reversing;
- **Flow Control:** Conditional and stochastic evaluation, flexible flow sequencing, amount filtering, type filtering;
- **Debugging:** Logging messages, measuring execution times;
- **Dynamic Loading:** To support exception encoding, port aggregation, etc. (see Section 3.8);
- **Persistence:** Storing and loading from the content pool;
- **Messaging:** Broadcasting, waiting and receiving messages.

5.1.1 Geometry 3D

The Geometry3D is the first and most developed procedure library which focuses on the generic creation, manipulation and analysis of geometric data. Up to this point, this library has been used for the generation of many urban structures, such as buildings (both indoor and outdoor), roads, trees and furniture.

The main entity type, defined and manipulated in this library is a B-rep geometry data structure called *Shape*, which, in turn, aggregates sets of other sub-entities: *vertices*, *edges* and *faces* (see Figure 5.2).

A vertex is an entity that holds a position in 3D space. It can also hold additional properties such as color, normal, texture coordinate, etc.

An edge is a structure that holds references to two vertices, connecting them. When used within faces, it contains a list of references to all connected faces (therefore allowing the definition of non-manifold meshes). It can also hold additional properties, such as color, tags, etc.

A face is a structure that is defined by a sequence of vertices, defining its closed, circular boundary. It represents a planar polygon, holding information of the normal of the plane that it lies on. A face can also incorporate additional vertex sub-sequences (also closed and circular), which define its holes.

The boundary representation with faces does not include an explicit representation of edges, but of half-vertices. A Half-Vertex is a structure that contains the information of a vertex relative to a certain face. Each vertex contains a list of *half-vertices*. Each half-vertex has a reference to the face that it belongs to, as well as its index in the sequence of vertices. It also holds properties such as color, normal, texture coordinate, etc. Each half-vertex is unique to a face and vertex.

Along with this geometric information, the shape entity features a 3D, box-like scope that, by including translation, orientation and size information, it operates as a local reference system.

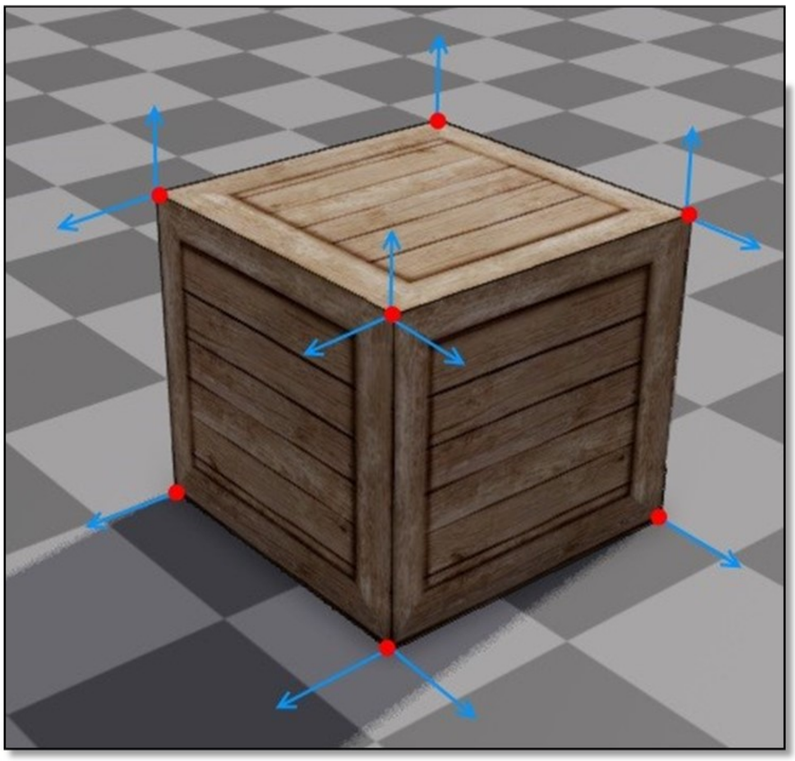


Figure 5.2: Box Shape with its vertices and half-vertex normals highlighted.

Such scope definition is essential for several advanced manipulation operations [M+06], such as relative splitting, relative translation, ray casting, etc.

This library also contains a definition of materials, which indicate properties for geometry visualization and rendering. Materials are defined per geometry entity, meaning that each face, edge and vertex may have its own.

Finally, this library supports a large number of manipulation procedures, which can be categorized in the following way:

- **Primitive Creation:** Generation of rectangles, circles, boxes, spheres, etc.;
- **Geometric Transforms:** Translation, rotation, scaling;
- **Geometric Manipulation:** Extrusion, offset, lathe;
- **Geometric Splitting and Decomposition:** Split through plane, split shapes into independent vertices, edges or faces;
- **Scope Manipulation:** Rotation, alignment to axis, to specific face or edge etc.;
- **Material Application and Texture:** Map texture to plane, box, cylinder;
- **Distribution:** Random acattering, placement in a grid-like structure;

- **IO:** GIS database loading, OBJ import/export, etc.;
- **Merging and Grouping:** To join faces and vertices into the same entity.

5.1.2 Streets

The Streets procedure library focuses on the creation, manipulation and analysis of street and road networks, but also generically applicable to any type of path. It features a simple representation of the medial axis of the streets and their crossings, as commonly characterized in GIS systems.

The main entity type, named *Street Network*, is composed by street vertices (or nodes), connected by street edges (see Figure 5.3). The nodes represent street beginnings/endings, crossings, intersections or simply control points to define curves. They can hold additional information, such as names, nature, crossing type, etc. Edges, on the other hand, correspond to the actual street segments and contain information about width, type, direction, lane number, condition, etc. A street, identified by name, is a sequence of segments, but no explicit structure has been designed to represent them. Finding streets implies simply the search for connected segments with segments featuring the same name.

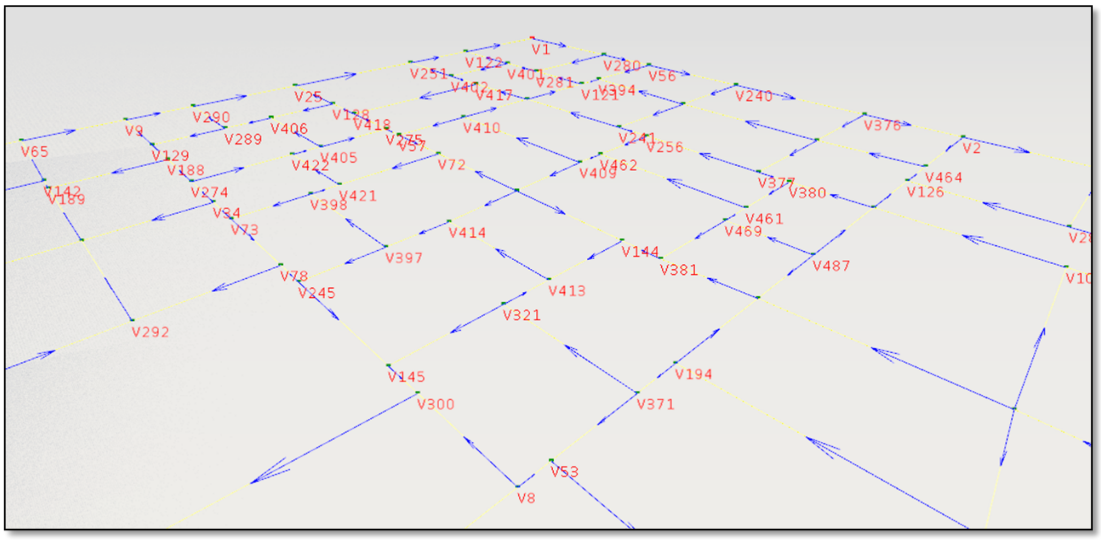


Figure 5.3: A Street Network entity.

This library includes a considerable number of manipulation procedures, which can be categorized in the following way:

- **Creation:** Definition of node sequences;
- **Property Management:** Setting tags, road names, types, etc.
- **Transformation:** Translation, rotation, increasing or decreasing precision, etc.
- **Merging:** Blending, detecting street crossings, etc.

- **Geometry Generation:** Buffering line segments to create street Shapes with sidewalks.
- **IO:** GIS, Openstreetmaps and database loading.

5.1.3 Surface

The Surface procedure library is dedicated to the generation of terrains, built upon the *Surface* entity. It is composed by a regular grid of equidistant vertices, stored in an array (see Figure 5.4). Each vertex carries a height value, relative to the surface base, but can also hold additional information, such as color, texture coordinate, material, etc. Because of its regular arrangement, explicit storage of (x, y) coordinates can be avoided, reducing the memory footprint. On the other hand, it simplifies the calculation of the height at any point of the surface.

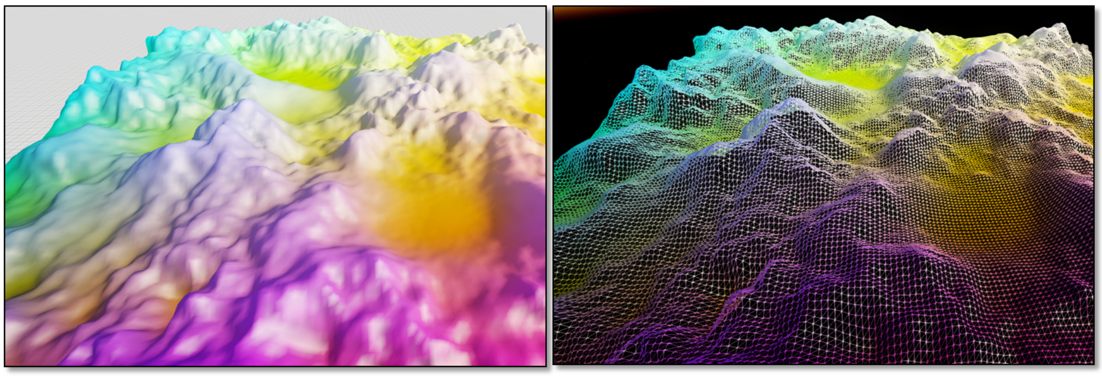


Figure 5.4: A Surface entity rendered with a basic shader (left) and a wireframe one (right).

This library defines a considerable number of manipulation procedures, which can be categorized in the following way:

- **Generation:** Perlin, fractals or diamond-square;
- **Transformation:** Translation, scaling;
- **Material:** Texture mapping, coloring;
- **IO:** Loading from images or point cloud data;
- **Placement:** Integrating and adjusting Shapes and Streets on the surface.

5.1.4 Text

The Text library is focused on text generation, but exceptionally does not operate on a specific entity, but instead simply provides functions that can be called within aspect expressions to produce values of string type.

This library includes several generator functions, which can be categorized in the following way:

- **Names:** First Names, Family Names;
- **Internet:** Domain names, e-mails, Usernames;
- **Addresses:** City, Streets, Zip codes;
- **Phone:** Phone numbers, extensions;
- **Lorem Ipsum:** Space-filling paragraphs, sentences and words.

5.1.5 Image Processing

The Image Processing library is focused on procedural image generation, built upon the *Image* entity, which defines a bidimensional array of RGB color pixels. It integrates with the Surface entity as the source for heightmaps, and integrates with the Shape, as a source for several types of material maps (diffuse, specular, bump, etc.).

Several manipulation procedures have been developed, which can be categorized in the following way:

- **Generation:** Pattern-based, Noise-based, Color Ramp, etc.;
- **Filters:** Blur, Pixelation, Sepia, etc.;
- **Transformation:** Cropping, Resizing, Rotation, Inversion, etc.;
- **Combination:** Comparing, Adding, Subtracting, etc.;
- **IO:** Obtaining images Webcam, Printscreen, Databases or File System.

5.1.6 Video

The Video library is a simple library dedicated to the generation of motion pictures. The main entity type, video, is a simple composition of image entities that, when shown in sequence and at a certain framerate, create the illusion of movement.

This library defines a small number of manipulation procedures, which can be categorized in the following way:

- **Joining:** Assembling sequence of images together at a certain framerate and quality;
- **Image-based:** Same as for the previous library, since a video is represented as a list of images;

- **IO:** Saving and loading video to disk, database or service.

5.1.7 Voxel

The Voxel library is dedicated to the development of voxel-based structures. The main entity type, the *Voxelchunk* consists of a three-dimensional array of voxel data, which, in turn, incorporate particular materials. Voxels are themselves not entities, as it would result in a large (and unnecessary) memory footprint. Like the Surface entity, the VoxelChunk contains an indication of its 3D offset within the world. However, it is oriented to the world axes and cannot be rotated or scaled.

This library includes a considerable number of manipulation procedures, which can be categorized in the following way:

- **Primitive Creation:** Generation of cubes, pyramids, spheres, etc.;
- **Geometric Transforms:** Translation, scaling;
- **Boolean Operations:** Union, Subtraction;
- **Material Application:** Changing voxel materials and colors;
- **Splitting and Decomposition:** Split through planes and decompose by material;
- **Manipulation:** Extrusion and offsetting;
- **IO:** Saving and loading the array to disk, database or service.

5.1.8 Game Entities

The Game Entities library concerns the creation, edition and analysis of game and other application-specific objects, that, aside from a visual representation, may encompass additional behavioral modules, whose properties have to be adapted to the environment where the object is being generated. The main entity type, the *Game Object*, comprises a list of entities of another type - the *Game Component*. This library follows therefore a component-based architectural pattern, making the construction of complex objects more decoupled, flexible and easy to manage.

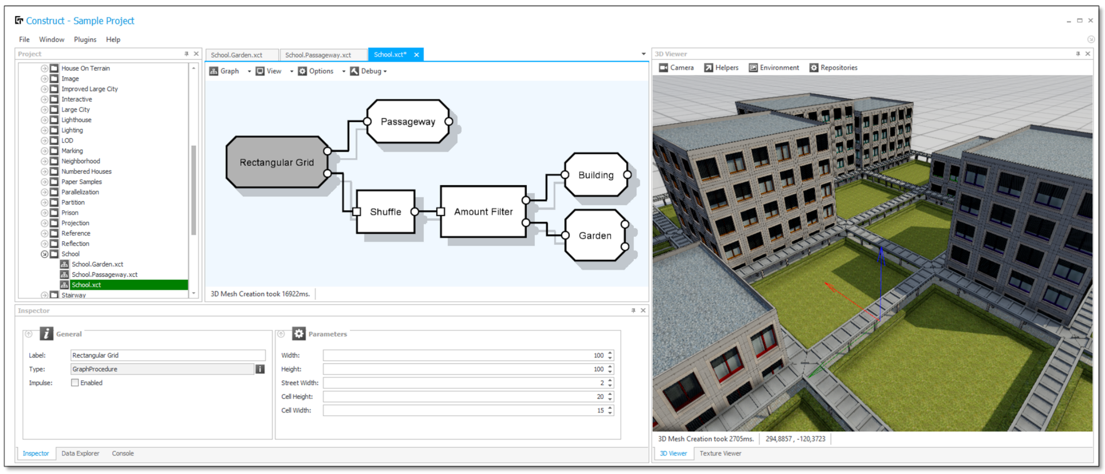
Several manipulation procedures have been developed, which can be categorized in the following way:

- **Instantiation:** Creating new components with certain properties
- **Joining:** Association or Composition of components into the gameobject

5.2 Visual Interface

The *Visual Interface* or *Visual Editor* has been built as a Rich Client Platform, which is capable of dynamically loading additional modules, commonly named *Editor Plug-ins*. These can operate in an autonomous and self-contained way or interact with other plug-ins, if necessary. During the initialization process, assemblies that have been imbued with the necessary metadata are loaded automatically.

There is a separation of Procedure Libraries and Editor Plug-ins. Procedure libraries contain only definitions of entities, aspect types and procedure definitions, being independent of the visual editor. Although they are the working base for the editor, they are meant to be used within other systems as well.



3D Mesh Creation took 16922ms.

Label: Rectangular Grid
Type: GraphProcedure
Impulse: ☐ Enabled

Width: 100
Height: 100
Street Width: 2
Cell Height: 20
Cell Width: 15

3D Mesh Creation took 2705ms. 294,8857 , -120,3723

Figure 5.5: Visual interface of the Construct Visual Editor.

The interface is composed by several dockable windows, whose layout can be freely manipulated and persisted. There is also a central *document area*, reserved for the display and edition of file content (see Figure 5.5).

The operation on the Visual Editor is based on the concept of *projects*, which is popular in many code and design tools. A project is a logical aggregation of files and folders - the resources on which users work. By default, the visual editor features a explorer window, where these resources can be viewed, organized and accessed in a tree-based structure.

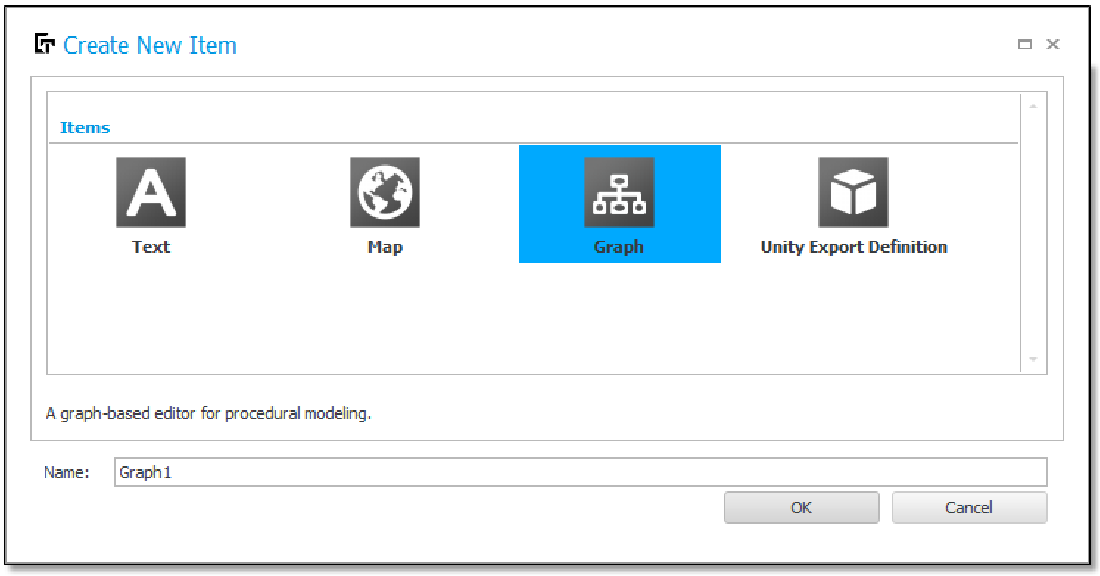
Another window already included in the core version of the Rich Client Platform is the *Inspector* window, which centralizes the display and edition of properties and options from files (Section 5.2.1), content viewers (Section 5.2.3), debugging options (Section 3.8.4) and graphs (Section 5.2.2).

The Visual Interface makes use of the Construct Framework's messaging engine to broadcast messages between modules, promoting plug-in decoupling and extensibility. For instance, when a file is created, opened, closed or deleted, corresponding event messages are broadcast, from the project editor, to all subscribed plug-ins, which can decide to perform its own action, such as refresh some resource cache or display some inspection options. Other examples of message broadcasts will be enunciated in further sections.

5.2.1 File Viewers and Editors

As a Rich Client Platform, the visual editor supports the association of specific file extensions to a given viewer or editor to display or edit its contents.

Files with associated viewers can be added to the project tree and opened, so as to view its contents. For example, the image rendering control is used to view the content of jpeg, png or other image files, which are most commonly used as material textures or surface heightmaps. The media player, on the other hand, is used to play effects, music and video files, which are typically used as source material for more complex media generation and/or as ambient sound in virtual environments.



A graph-based editor for procedural modeling.

Name: Graph1

Figure 5.6: Creating a new file.

Files with associated editors, on the other hand, cannot only be viewed, but also have their contents modified. Consequently, they also introduce the possibility of creating new files of the given extension (see Figure 5.6). The simplest illustration of a file editor is the one used for .txt or .xml, featuring text edition facilities.

Uncontemplated file extensions can still be added to project folders, even though no default editing or viewing action on them will be possible. However, they might represent sources loadable by graph nodes (e.g. esri .shp files, .fbx files) or even a dependency of another file (e.g. a .mtl material file of an .obj mesh file).

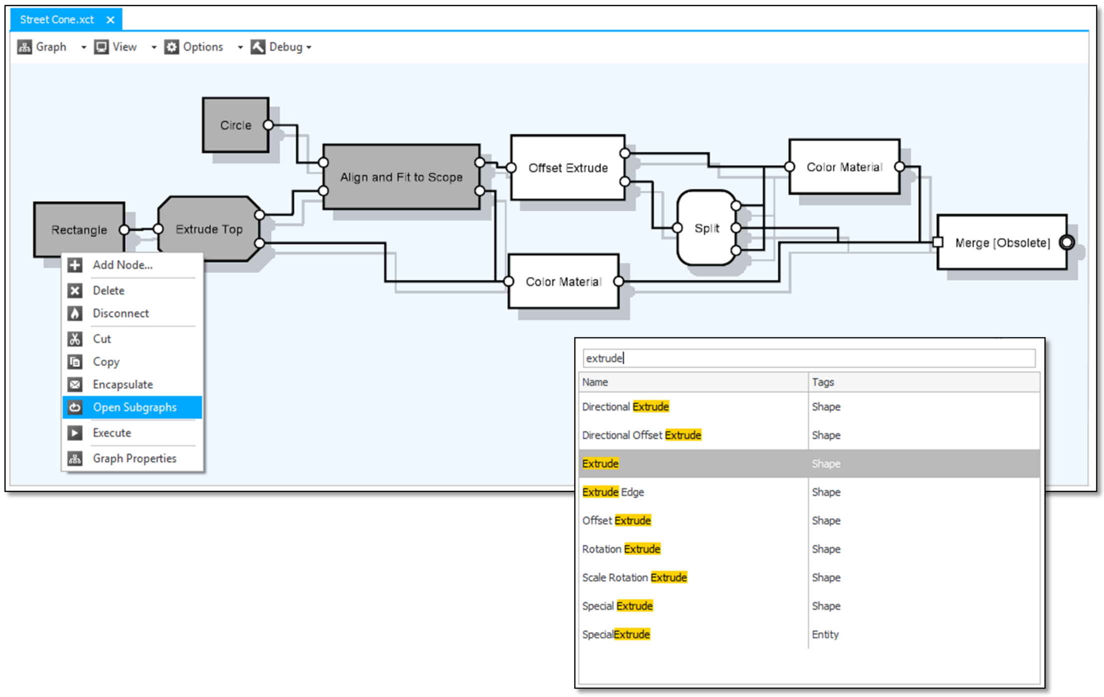
File editors and viewers are displayed in the *document area* of the Construct editor, organized as a set of tabs. It is possible to have several tabs featuring the same type of editor, but only one instance per file.

5.2.2 Graph Editor

The graph editor is one of the implemented file editors, which focuses on the manipulation of *.xct* files, the chosen files for storing graph definitions. As mentioned, graphs are stored in XML format, and this very same format is used for copying and pasting graph excerpts within or across graphs.

The graph editor is composed by a canvas, where the nodes, ports and edges are drawn (see Figure 5.7). Implemented using the Microsoft XNA Framework, it is able to handle large graphs with ease.

Node, port, edge and graph configuration is centralized on the mentioned inspector window. When a node is selected, the user can edit its parameters, attributes and augmentations. Parameters can have specific controls (textboxes, combobox, sliders), depending on their type and assigned constraints, and can be switched to expression parsers, featuring the syntax presented in section 3.4.1. When a port is selected, the user can change its state to "blocked" or to "gate", as well as define its name. As for the graph options, the user is presented with some metadata editors (name, author, description, etc.) and with appropriate list controls to add and configure graph aspects.



extrude

Name	Tags
Directional Extrude	Shape
Directional Offset Extrude	Shape
Extrude	Shape
Extrude Edge	Shape
Offset Extrude	Shape
Rotation Extrude	Shape
Scale Rotation Extrude	Shape
Special Extrude	Shape
SpecialExtrude	Entity

Figure 5.7: Graph Editor Canvas with quick search window and with highlighted node selection ready to be encapsulated.

A couple of facilitators for graph manipulation have been introduced. The insertion of nodes is possible through a quick search window that lists all available procedures. Once the node is added, edges can be drawn by dragging the mouse from one port to another. Since ports can be of different types, visual guides on ports have been introduced (changing their color and size) to clarify what types are compatible and which connections are allowed.

Execution of the graph has, by default, to be explicitly triggered by the user. Since the whole graph is executed on a different thread, interaction remains smooth during the process. When the execution is finished, all content is gathered and broadcast, as a specific message, to all subscribed visual interface modules, such as the Content Viewer (see Section 5.2.3) and the Data Explorer (see section 3.8.4).

The implemented *Live Execution* mode will automatically re-execute the graph for every change. This is not only convenient for a continuous operation, but also helps the user understand the impact of modifications on the graph result. However, when building longer and more complex graphs, execution can take longer times, introducing a delay between the graph change and the demonstration of its result. In such situations, this option is best left off.

The encapsulation of graphs has also been made easy to achieve. To include a graph in a supergraph, the user simply has to drag the graph's file into the supergraph's editor canvas and the encapsulated node will be instantiated. The user can still change the subgraph's parameters and gates afterwards and the changes will be reflected on the supergraph immediately. Alternatively, a user can select a subset of nodes and edges and choose the option to encapsulate that selection into a new graph file. Referenced parameters/attributes and connected ports will automatically be configured and added to the encapsulated node's signature.

To facilitate experimentation, edges can be toggled on or off in a similar way one would comment programming code sections. When deactivated, the existence of the edge will be ignored, providing some insight over the effect of that connection (or of the following node sequences) towards the generated graph results.

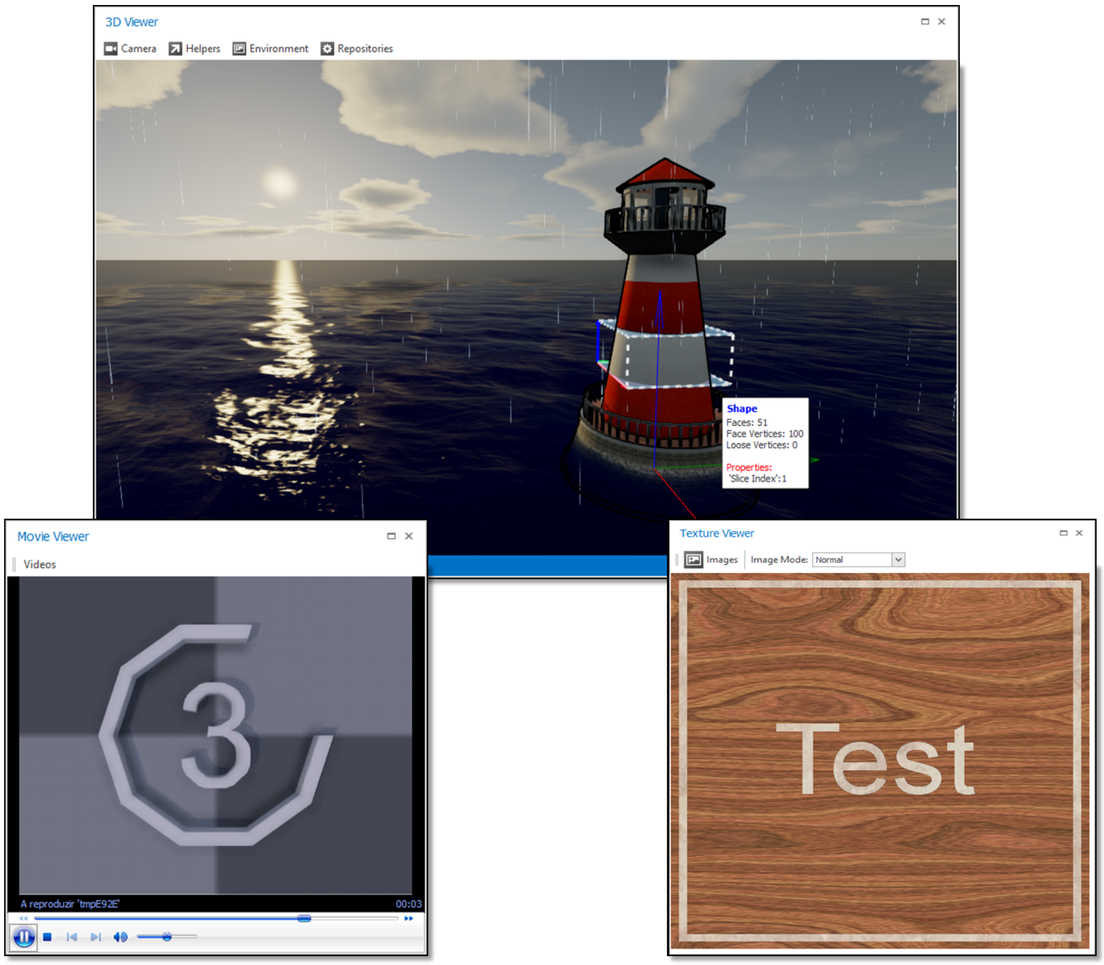
A robust recovery mechanism has been included in the graph editor. During the parsing process of the graph XML, references of low-level or encapsulated procedures are recursively loaded by their namespace or location, respectively. If procedures would need to have their names, namespaces or locations changed, the loading process would fail due to broken references. In such cases, the unique identifier (see section 5.1) is used by the recovery mechanism to search for the procedure and restore the reference.

5.2.3 Content Viewers

One of the main purposes of the Construct editor is allow immediate content visualization and evaluation after the execution of procedural content graphs. Since the produced data can be composed of several entity types, diverse types of visualization tools may be required as well.

The rendering of 3D geometry is assured by a another XNA-based control, featuring the DigitalRune Framework, which adds several higher-level game objects, a physics engine and some rendering facilities, such as deferred rendering and shadow mapping.

To view the graph-generated result, an XNA-based, 3D rendering window is available (see Figure 5.8). Here, a couple of view options have been introduced. This includes a geometry picking mechanism that allows a better perception of shape scopes (see section 5.1.1), an explicit rendering of face edges, post-processing effects, among other guides. Also, several environmental details can be toggled and controlled, such as ocean water and a daylight system, which enable a better perception and embodiment of the generated objects.



Shape
Faces: 51
Face Vertices: 100
Loose Vertices: 0
Properties:
'Slice Index':1

Figure 5.8: Texture, Movie and 3D Rendering Windows.

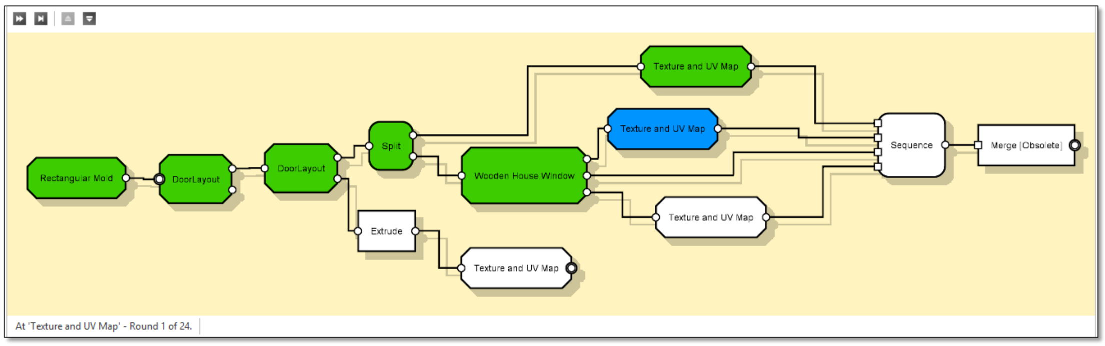
For the visualization of procedurally generated textures and media, simple image and movie viewers has been implemented (see Figure 5.8). Since many entities of these types can be generated at once, each of these windows features a simple gallery for the navigation through all the generated individual images/movies.

5.2.4 Debugging

As in any procedural-based environment, the flow of data and exact sequence of operations may be difficult to predict and control. As the result of a graph may contradict the user expectations, the possibilities of debugging are essential to understand the point of failure and the necessary steps to fix it. Otherwise, development may become overly strenuous, complex and time-consuming.

The simplest means of debugging lies on the use of the *console* window, where all textual log information is printed to. Again, logging is enabled through the reception of specific messages, which can emanate from editor plug-ins or even procedure libraries. The "Log" procedure, for instance, features a control parameter to which fixed values, global parameters, global attributes or expressions can be associated, evaluated, and then forwarded as log messages. This provides some insight of the data flow and graph status at the particular graph locations where this procedure is placed.

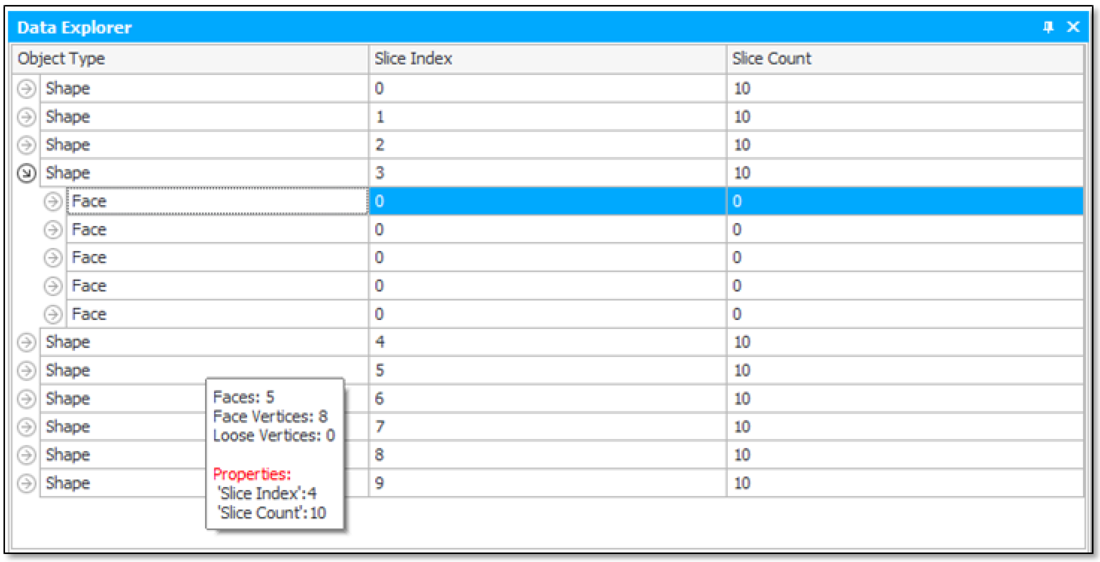
Another, more advanced debugging feature is the *step-by-step debugger*, which is more deeply integrated with the graph editor and works as an alternate graph execution mode. When activated, a graph viewer (similar to the standard graph editor) is opened and the execution process starts, but blocking at the first round of the first node. From here, the user can choose to proceed stepwise to the execution of next rounds or nodes, which are highlighted with different colors as the activity progresses (see Figure 5.9). Also, at each step of the execution, a copy of all the entity data queued at all the input and output ports of all nodes is collected and broadcast, being shown on the respective content viewers. Thus, the user can view the evolution of the content data at throughout the graph execution and perceive erroneous behaviors.



At 'Texture and UV Map' - Round 1 of 24.

Figure 5.9: The Step-by-Step Debugger.

Finally, the *data explorer* is a plug-in window that, subscribing to content broadcasting messages, collects the generated entities and lists them on a hierarchical tree control. The purpose of this tool is two-fold (see Figure 5.10). On the one hand, it allows the user to view and understand entity composition, i.e. what entities and sub-entities exist and how they are stacked (such as the Shape having faces and vertices). On the other hand, it provides insight of attributes values of each entity. The attributes of the current graph (which are known, since they are specified at the graph properties) are displayed on columns, building a matrix of entity-value correspondence. When used together with the step-by-step debugger, the data explorer offers a more complete analysis of the graph data flow. This is especially helpful to understand side effects and figure out unwanted results.



Data Explorer

Object Type	Slice Index	Slice Count
Shape	0	10
Shape	1	10
Shape	2	10
Shape	3	10
Face	0	0
Face	0	0
Face	0	0
Face	0	0
Face	0	0
Shape	4	10
Shape	5	10
Shape	6	10
Shape	7	10
Shape	8	10
Shape	9	10

Faces: 5
Face Vertices: 8
Loose Vertices: 0

Properties:
'Slice Index':4
'Slice Count':10

Figure 5.10: The Data Explorer.

5.3 Connection and Incorporation

One of the main objectives behind the development of the Construct has always been to make it open, adaptive and connectable to other systems. The flexibility of its architecture has introduced this opportunity in two ways:

- **Connection**: By allowing the construction of custom import and export procedures and by enforcing a methodology based on the co-existence and combination of different entity types;
- **Incorporation**: By allowing the execution of graphs and the use of procedure libraries within environments other than the visual editor.

The possibility to import and export data is common feature in any content development system. The usual strategy consists in building functions that load known data formats and convert them to the data structures supported by the system, so that they can be further worked on. Once the content is finished, it can later be converted back to another data format. However, both the number of supported types and import/export formats is often restricted to a limited number of types and popular standards. For instance, there are no standards for representing voxels structures or game entities with behaviors. This limits the possibilities to develop and integrate particular types of content.

This gap is addressed in the Construct system, as it provides an extensible Application Programming Interface (API), where new types of entities, aspects and procedures can be developed and manipulated. New procedures can also pose new ways of persisting and conveying the content to other systems, such as for importing/exporting custom file formats or for delivering the data via sockets or web requests (see Figure 5.11). On the other hand, the visual editor can be enhanced with plug-in viewers and editors to facilitate data handling or to provide alternative visual interfaces to advanced connectivity options (see Figure 5.12).

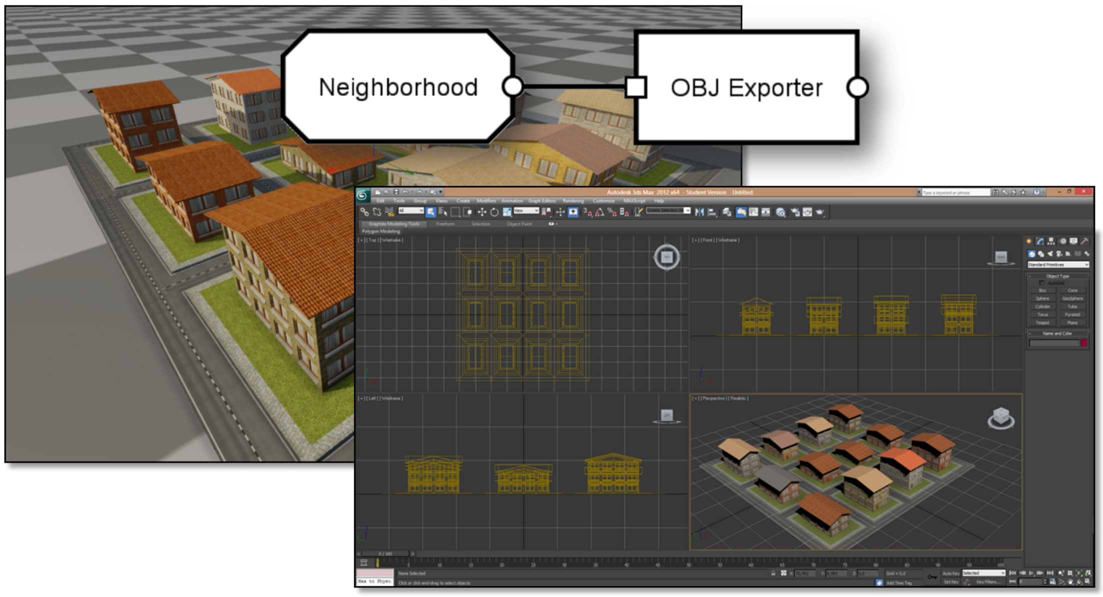


Figure 5.11: Export of geometric data using custom procedures.

Another issue with standard development systems regarding its connectivity lies on the construction of a pipeline that links various development systems. In practice, importing and exporting data are actions that require an extra step beyond the normal design process. In other words, exporting is an explicit operation that must be executed every time the content is changed. Importing requires, in most cases, the user to discard the manual labor already executed upon the previous data. In the Construct's procedural, graph-based approach, the process of import/export of data can be integrated seamlessly into the design process, improving the connectivity possibilities (see Figure 5.11).

On the other hand, incorporation is a feature that makes particular sense in the scope of procedural generations systems. For instance, by allowing the execution of procedural processes inside a game, training or simulation engine, different content can be produced on each run. As a result, many (if not infinite) experiences can be held without going through the tedious task of exporting and persisting the generated content. The Construct framework introduces this possibility in its Core library, which contain graph loading and execution methods that can be called within any system supporting .NET assemblies. Such is the case of game engines such as XNA, Mogre, Neoaxis or Unity. As a result, they can be used not only within desktop, but also mobile applications. At this point, the Unity incorporation has been successfully implemented (see Figure 5.12).

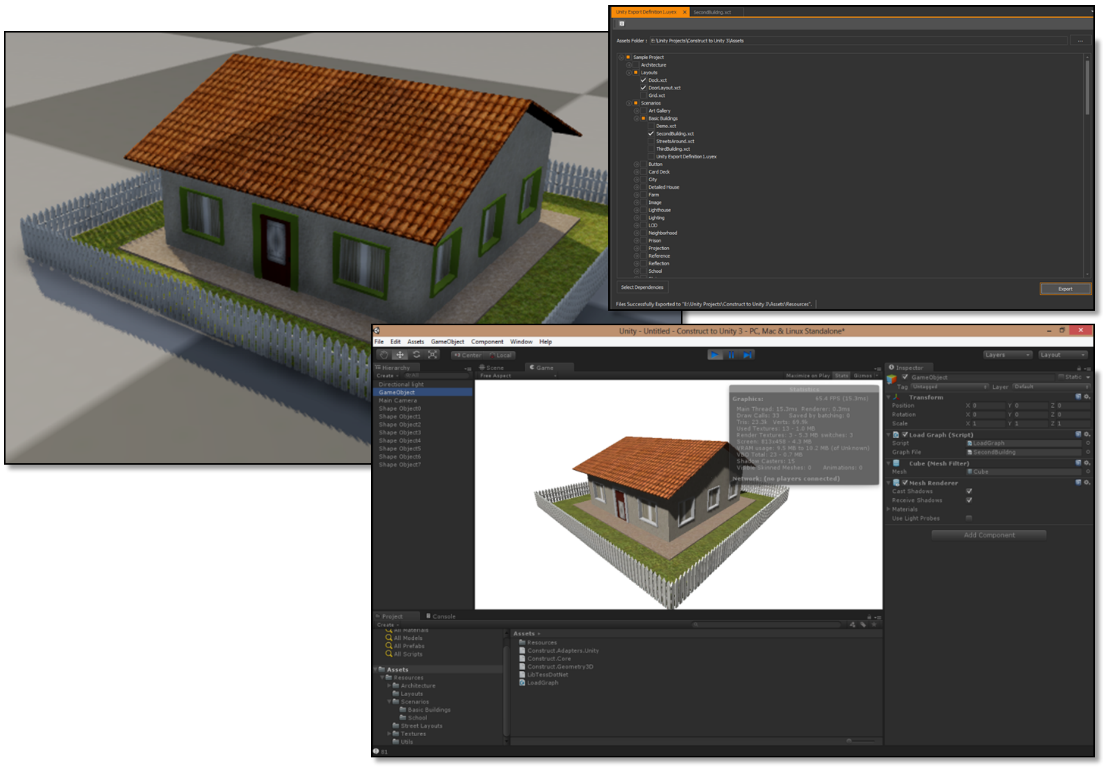


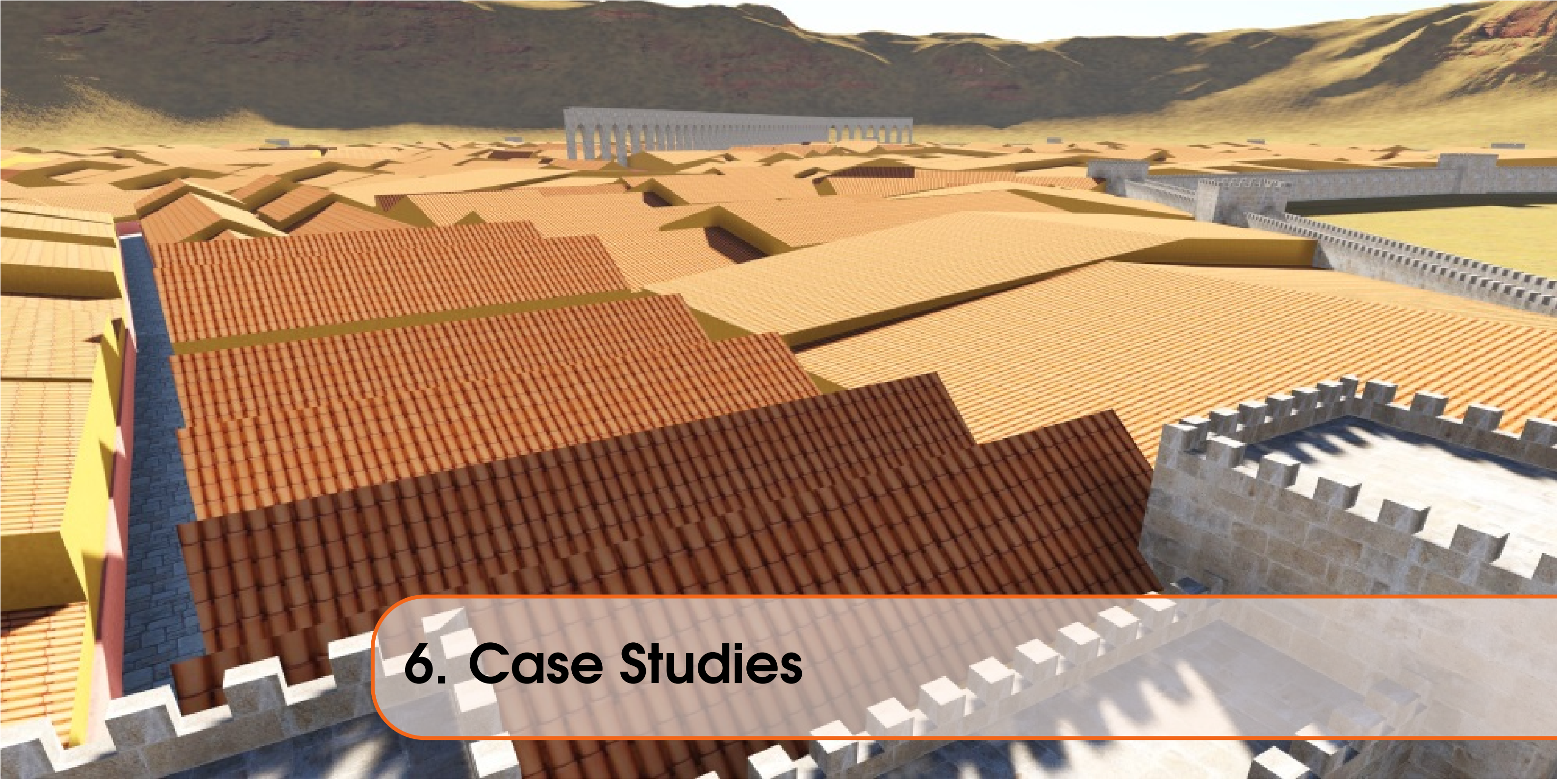
Figure 5.12: Exporting graphs and assets from the Construct visual editor, so that they can be used and executed directly inside the Unity editor and derived projects.

When connecting or incorporating processes, there is always the issue of finding a common ground, a translation of data types known by each system. This need is a consequence from the natural differences between tools, methodologies and technologies, but also from their purposes. An *editable Mesh* in the popular tool Autodesk 3Ds Max is processed as a *Shape* in the Construct and as a *GameObject* with a *Mesh Filter Component* in Unity. Each has its own distinct structure and uses, but there is a common ground between them - they all represent 3D geometries. In the Construct, establishing a mapping of one system's entity to another is called *realization* or *materialization* and can be configured *a priori* to facilitate a seamless integration between systems.

This process occurs in the 3D Rendering of the visual editor itself, organized in the form of *realizers*, which perform one-to-one, many-to-one or one-to-many correspondences. The first case applies, for instance, for surfaces, since each is materialized into a single vertex buffer. As for shapes, which can assume very small proportions, it is often more advisable to aggregate them by material/shader so as to improve rendering performance, in which case the many-to-one correspondence makes sense. On the other hand, if a face/edge/vertex picking feature is intended, a one-to-many conversion is required. The realization option can be changed at any point.

5.4 Summary

With the purpose of experimenting and validating the proposed methodology, a generic prototype system for procedural content generation was developed and explained in this section. The *Construct*, as it was named, consists of a framework and a visual editor. The first is composed by several procedure libraries for the definition of entities, aspects and procedures, while the second consists of several plug-ins for the specification, manipulation, visualization and export of procedural generation processes, including, but not limited to, procedural content graphs. Both parts were built with extensibility in mind and conceived to be open, adaptive and connectable to other systems, such as other content production tools and virtual platforms.



6. Case Studies

Throughout the development of the current Ph.D. research, and as the implementation of the Construct became more mature, many possibilities for it to serve other uses and projects have arisen. This chapter focuses on describing such case studies in more detail, demonstrating in which way Procedural Content Graphs have promoted and facilitated the development of several new procedural generation approaches.

6.1 3DWikiU

3DWikiU – 3D Wiki for Urban Environments (PTDC/EIA-EIA/108982/2008) is a research project, which took place from 2010 to 2012, funded by the National Foundation for Science and Technology (FCT), and led by researchers from INESC Porto, the Faculty of Engineering of the University of Porto and the University of Trás-os-Montes and Alto Douro.

The purpose of the project was to build a system that could manage an interactive and immersive virtual tridimensional environment incorporating a set of digital services. Being available to its users, the services would allow data sharing and collaborative work for further content improvement, this way promoting activities in urban planning, virtual tourism, public services, cultural and natural heritage, education or even commerce.

Concerning urban planning services, the goal was to obtain a reliable 3D replica of the real urban environment, whose base could be built procedurally from geographical data, so the users can navigate on it as they would in the real urban environment (see Figure 6.1). It was intended to offer collaborative edition, so the users can also modify the virtual world to improve visual fidelity (changing textures, adjusting building heights, settings urban furniture locations, etc) - a *3D Wiki of urban environments*.

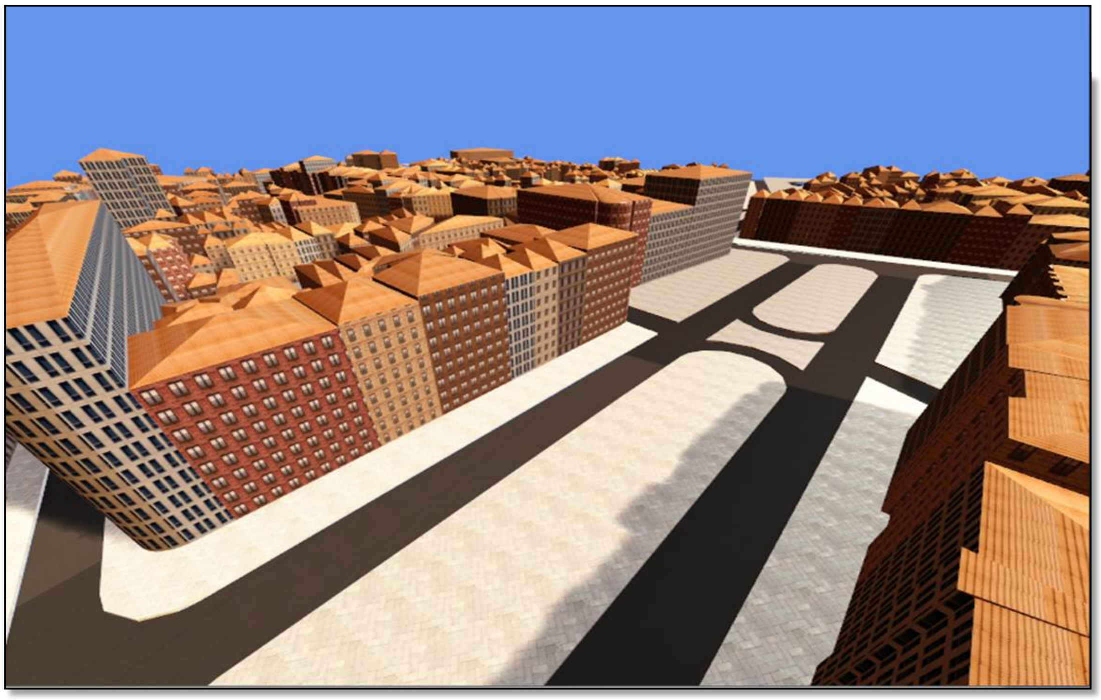
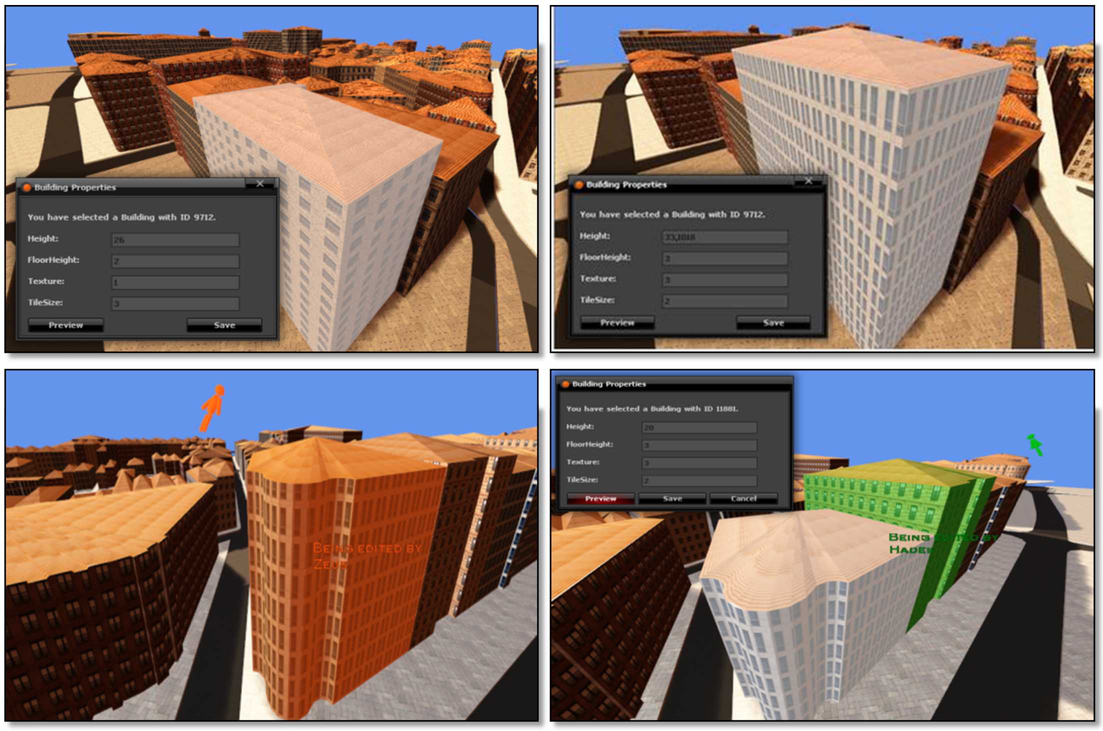


Figure 6.1: Overview of a section of Avenida dos Aliados, in the city of Porto, Portugal. Aside from some block, street and building layout information, as well as some some height data, no other real-world details were available to model the city with higher fidelity.

More specific purposes could be fulfilled, such as allowing a group of architects to work on a common project remotely and simultaneously, and introducing the possibility for their clients to give feedback directly on the project models. In a similar fashion to some municipal services, users could report situations (and propose solutions) that would need the attention of the local authorities. Regarding the virtual tourism field, the user would be able to discover the place that he would plan to visit and have some previous contact with the local culture, so he could plan his trips. He should also be capable of making reservations of the hotel services if they were available.

The system followed a client-server architecture model. The server, acting as a central hub for all incoming client communications, contemplated an urban ontology that was used to map multiple (local or remote) GIS databases into a single unified urban model [Mar+12]. This model contained parameterized information about the urban features, organized by their semantic relevance, spatial reference and feature type (e.g. buildings, windows, streets, trees). The purpose of this approach was to make it easier to handle and export such parametric data so as it could be used in further tools or studies. The server's purpose was also to manage user contributions to the urban model, by controlling their concurrent and asynchronous nature and by relaying the changes to all users.

The client application was the main "window" of the system, with which the users interacted. Visualization and input was handled on this side, which then communicated with the server. When a user visited a certain location on the client application, all the features surrounding


Building Properties

You have selected a Building with ID 9712.

Height: 26

FloorHeight: 2

Texture: 1

TileSize: 3

Building Properties

You have selected a Building with ID 9712.

Height: 33,1018

FloorHeight: 3

Texture: 3

TileSize: 2

Building Properties

You have selected a Building with ID 11881.

Height: 20

FloorHeight: 3

Texture: 3

TileSize: 2

Figure 6.2: A building being modified by changing its corresponding parameters. On the top left: the building when clicked the first time. On the top right: after the height, texture, number of floors and number of windows per floor have been changed and the building has been regenerated with the new parameter values. On the bottom left: Building being edited by user "Zeus", as seen by user "Hades". On the right: Own building edition view led by "Zeus", while user "Hades" is changing the neighboring building.

the area were requested to the server, which would then look it up in its unified data model. There, spatial queries were performed to obtain all the features of the indicated types. Once this parametric information was returned to the client, it was employed by Procedural Content Graphs to generate the actual 3D city models, with specific services and behaviors. The Construct Framework was therefore a ground stone that allowed such dynamic content production and adaptivity based on parametric data, making the overall data handling process much simpler to manage.

This approach intended to solve the various issues concerning bandwidth, memory and computing usage, user interaction usability and the integration with existing information. By default, a user would view the world from a first-person perspective, using the keyboard to move and the mouse to control the camera direction. When an entity, such as a building, was focused, a second metaphor of interaction was introduced: a windowed GUI was presented, containing the various parameters that could be changed (see Figure 6.2). When the parameters were changed, one could preview the effects and, when the intended result has been achieved, they can be stored and replicated through the server, as well as propagated to the other clients by simply updating some parameters. As in any collaboration system, the server provided a versioning system that would allow them to keep a history of all performed changes. This worked as a way to introduce undo and redo operations, but also to safeguard the system against possible vandalism.

6.2 ERAS

ERAS - Expeditious Reconstruction of Virtual Cultural Heritage Sites (PTDC/EIA-EIA/114868/2009) is a research project, which took place from 2011 to 2013, funded by the National Foundation for Science and Technology (FCT), and led by researchers from INESC Porto, the Faculty of Engineering of the University of Porto and the University of Trás-os-Montes and Alto Douro.

The purpose of ERAS was the development of an expeditious modeling system to reproduce archeological sites and ancient monuments. Starting with text extraction and natural language processing, this system operated automatically, oriented by knowledge of construction rules of such buildings and monuments. This system had as a starting point the coding of the architectonic processes of each historical period through formal grammars.

The selected case study were the Conímbriga ruins, for which extensive textual descriptions and CAD-format footprints of several buildings were available. The final model included procedurally generated buildings as well as manually modeled ones (see Figure 6.3).

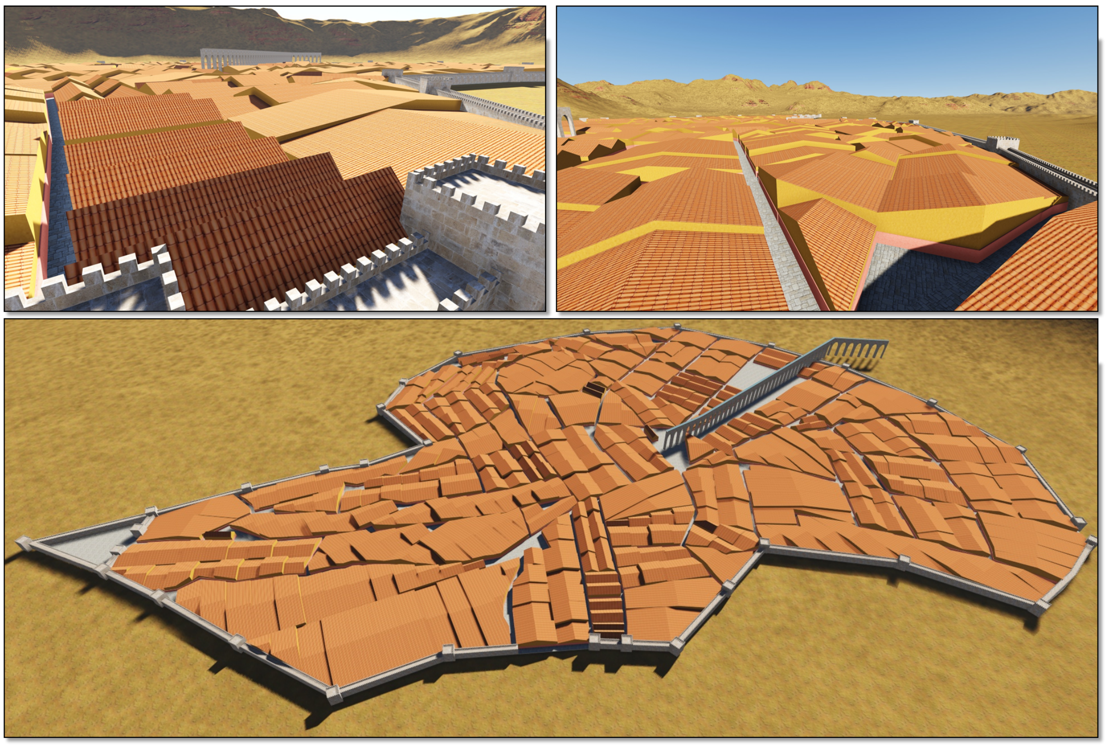


Figure 6.3: Procedural reproduction of the Conímbriga Roman ruins.

The automatic analysis and parsing of the textual sources had as purpose the extraction of parametric data, which would feed a shape grammar-based modeling technique, capable of producing both interior and exterior building sections [Aa+14]. This technique was employed in examples which followed a particular pattern structure, which could be encoded into such generation rules. Other descriptions were analyzed and used as inspiration for designers, who manually constructed a few other pivotal buildings. One of the purposes of the Construct system was to integrate both modeling paradigms within a single pipeline using procedural content graphs. The models, exported to a common geometric format (.OBJ) were then imported, positioned and adjusted, according to available georeferenced coordinates, into a single environment.

Given the scarce amount of available information, another use of the Construct System was to complete the Conímbriga ruins scenery with a set of buildings, streets and walls in the spaces for which no descriptions were available. Although no particular fidelity could be assured, this provided a plausible urban landscape by using similar architectonic characteristics, materials and layouts. The whole process was chained with the model integration step, so that the real model data could serve as a base with which the fictitious content could fit.

6.3 Parcel Generation

The Parcel Generation project is an Master thesis project taken by Lennart Kroes, a student from the Delft University of Technology and researcher at the TNO: Netherlands Organisation for Applied Scientific Research, from late 2013 until mid 2014. The purpose of the project [Kro14] was to develop a methodology for procedural generation of realistic residential parcels, and has been developed using the Construct System (see Figure 6.4).

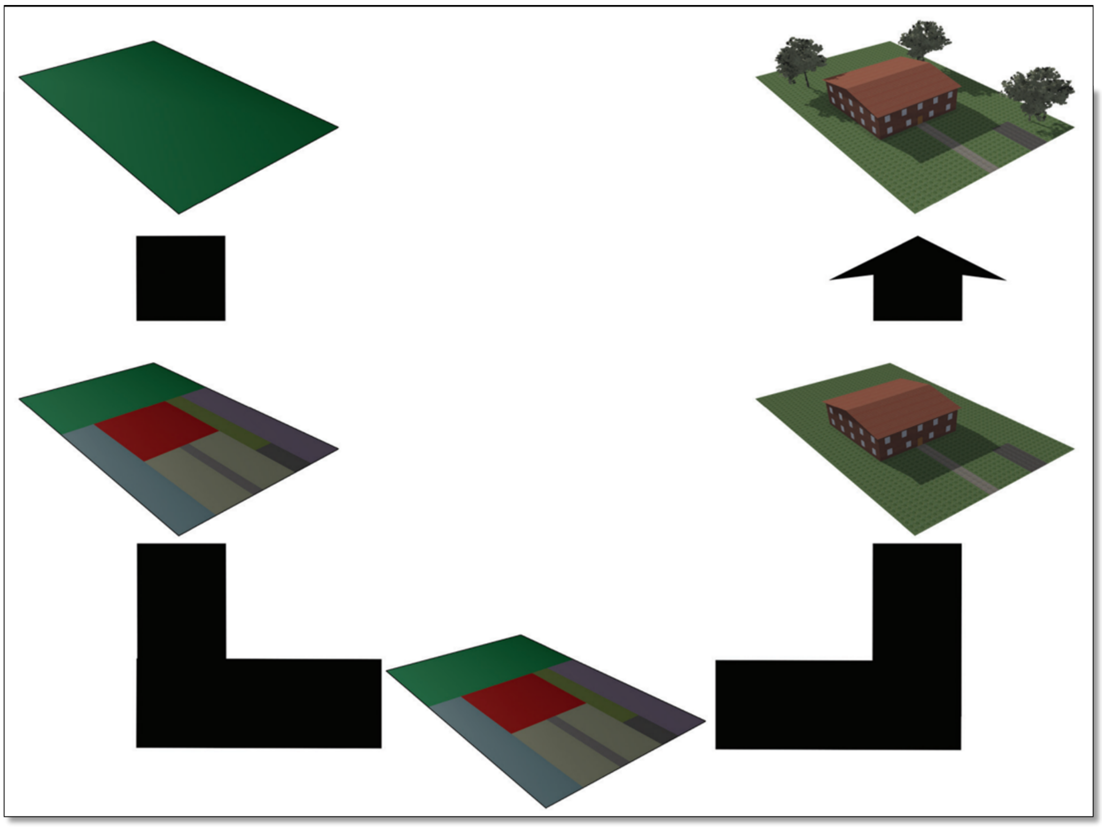


Figure 6.4: Parcel generation process in the scope of Lennart Kroes' work approach within the Construct [Kro14].

More concretely, the process consisted in dividing residential plots into zones (front yard, house footprint, backyard, patio, etc.) and fill these zones with objects (such as chairs, plants, trees, barn, etc.). Until this point, related work had only focused on a more simplistic and unrealistic generation of these plots, with configuration where the houses would be simply centered on the available space. However, urban guidelines are far more complex and varied, making them a target for attention which motivated this work.

The work consisted in combining two different generation approaches: procedural content graphs and layout solving techniques. The first introduced the opportunity to define parcel configurations with adaptive sizes and flexible variations, which could be encoded into reusable templates, all of which could having the particular benefit of generation performance. Layout solving techniques, on the other hand, were used for the automatic placement of objects, based on the definition of rules and relationships between these objects, as defined in a semantic library. Because this approach allowed objects to be managed in a more flexible manner, its the effectiveness was high, but at the expense of the efficiency of the method.

The implementation within the Construct was achieved in the form of a Construct procedure library, which included several new nodes that operated on a newly introduced *Zone* entity, a subclass of Shape (see Section 5.1.1). Also, several new GUI features were included to trigger and configure the parcel generation process. Once started, the process would create a moderator, which, in turn, would execute a selected procedural content graph and manage the exchange of its generated data with the layout solver. The final scene was then gathered and sent to the rendering window or exported.

6.4 Data Integration

The Data integration project was a summer project taken by Peter Heinen, a researcher at the TNO: Netherlands Organisation for Applied Scientific Research, around mid 2013. The objective was to create a set of building generation graphs for typical office, residential, and commercial buildings found in Oman (see Figure 6.5).

The input and inspiration for these graphs were freely available photographs. A series of new graph nodes were implemented and assembled into procedural content graphs, which were then batch executed from an independent console application that would make use of the Construct framework and its procedure libraries. Given a GIS areal vector shape layer with building footprints and building type and height attributes, the graphs would generate a corresponding set of building models in OpenFlight format, as well as a GIS point vector layer with model position and information about their orientation. The resulting models were then incorporated in the Havok simulation environment.

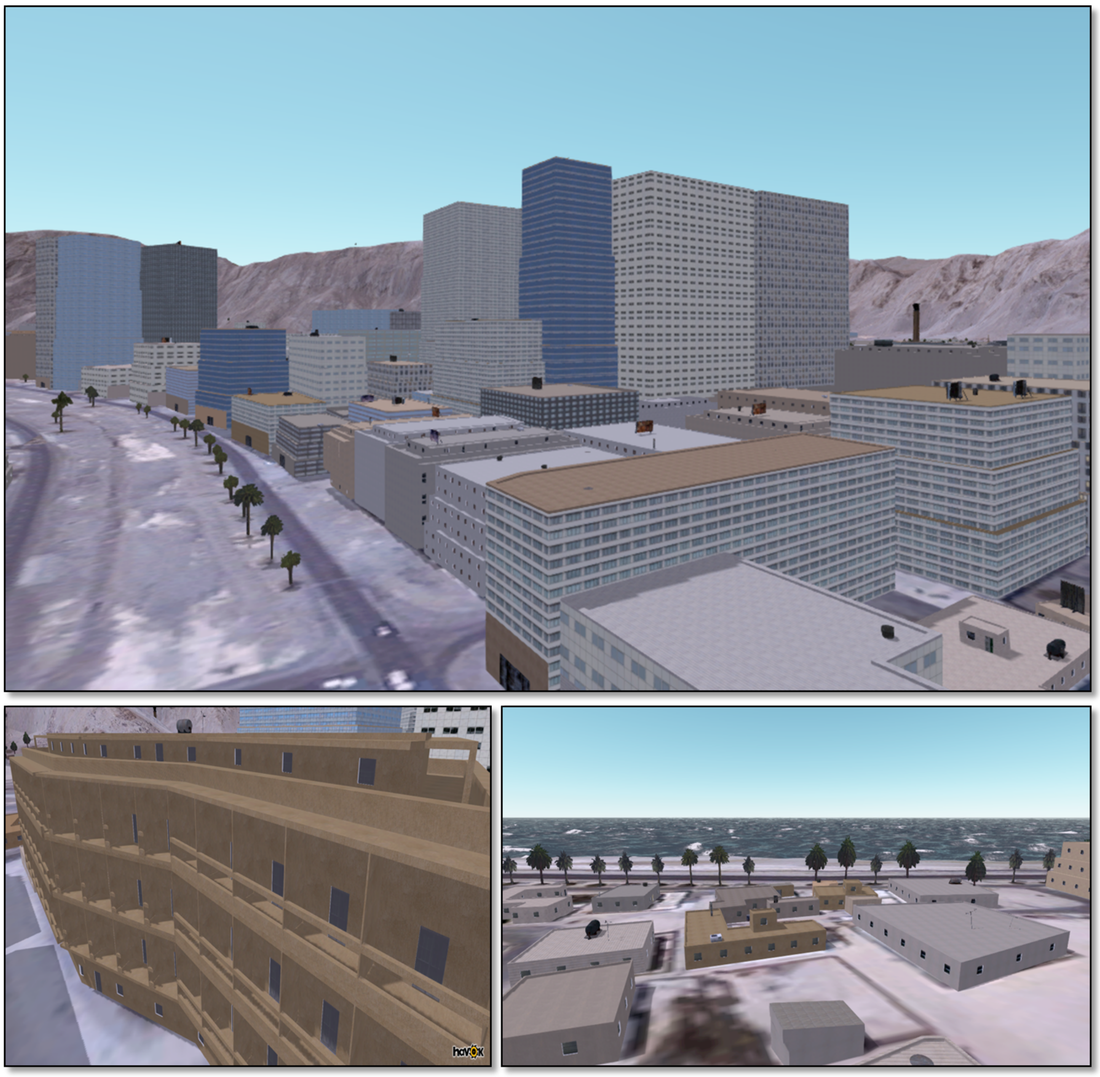


Figure 6.5: Oman city reproduction in the scope of Peter Heinen's work approach within the Construct.

6.5 Vessel Generation

The Vessel Generation project is an ongoing Master thesis project taken by Nathan Mol since early 2014, a student from the Delft University of Technology and researcher at TNO: Netherlands Organisation for Applied Scientific Research. The purpose of the project was to develop a methodology for procedural modeling and variation of nautical vehicles (see 6.6), having been fully developed using the Construct System.

The work consisted in the development of a procedure library featuring new nodes for the planning of weather deck layout, cargo layout management and placement, simple geometry generation (e.g. for crew accommodation), and texture production. This ease of implementation of custom procedures was highly praised by the student, not only for the increase of the generation opportunities, but also for the possibility to develop new testing and validation abilities. Contrary to shape grammar alternatives, PCGRs offered a structured and readable specification, while

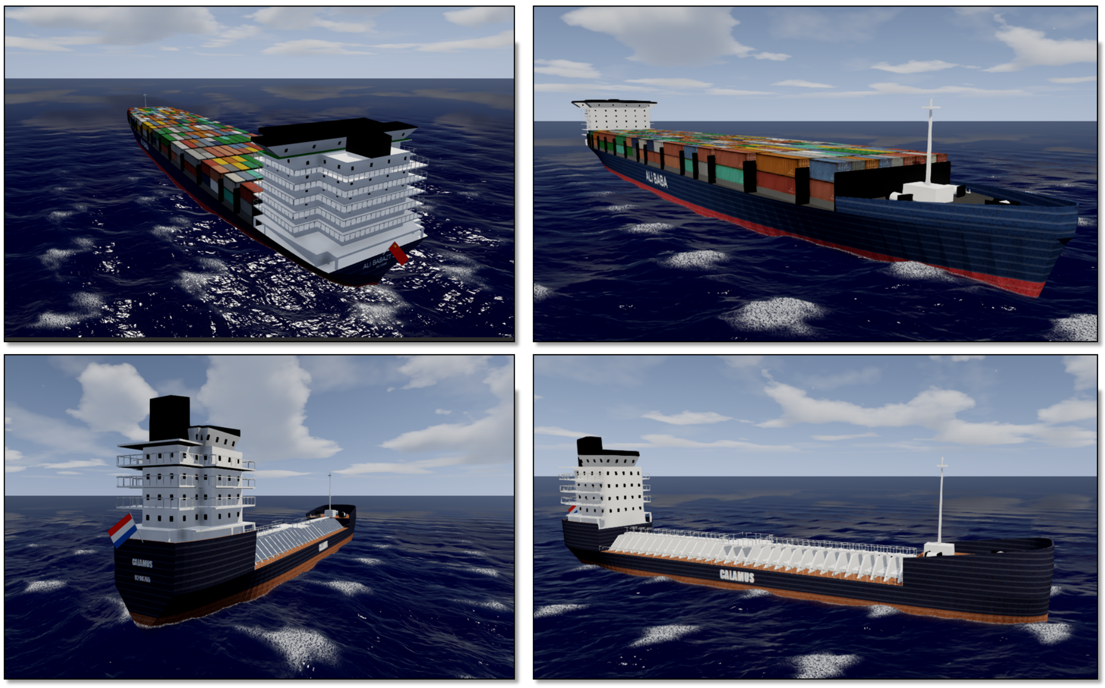


Figure 6.6: Vessels generated in the scope of Nathan Mol's work approach within the Construct.

graph encapsulation, global parameter and (local) attribute management improved the power of graph-based approach significantly.

Another used feature of Construct was the GUI component implementation of user-specified file extensions (see Figure 6.7). With this feature, it was possible and simple to create and save configured instances. It also allowed an easy design and adjustment of the base vessel hulls by modifying some parameters and major guides, facilitating the test and validation of the developed method.

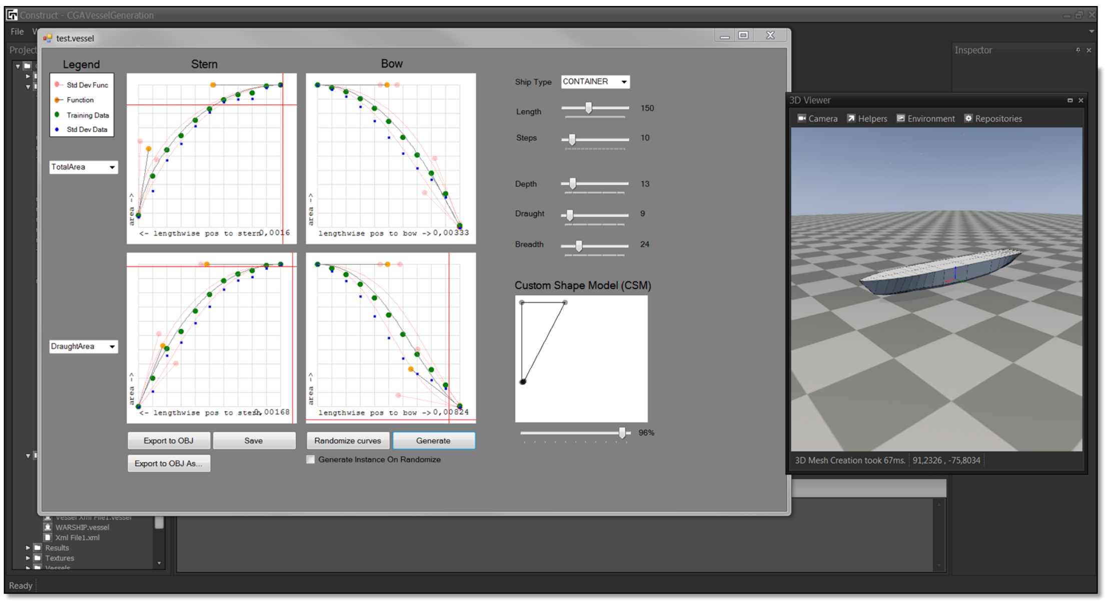


Figure 6.7: GUI component for the support of Vessel generation.

6.6 Ecosystem Generation

The Ecosystem generation project is an ongoing Master thesis project taken by Benny Onrust, a student from the Delft University of Technology, since mid 2014. The objective of the project is to develop a methodology for the procedural generation of natural environments using ecological models.

Part of this project consists of creating a real-time 3D application for the visualization of natural landscapes within web browsers. The Construct is used for the generation of the 3D plant models using a L-system approach, as well as to export them to be used in other applications.

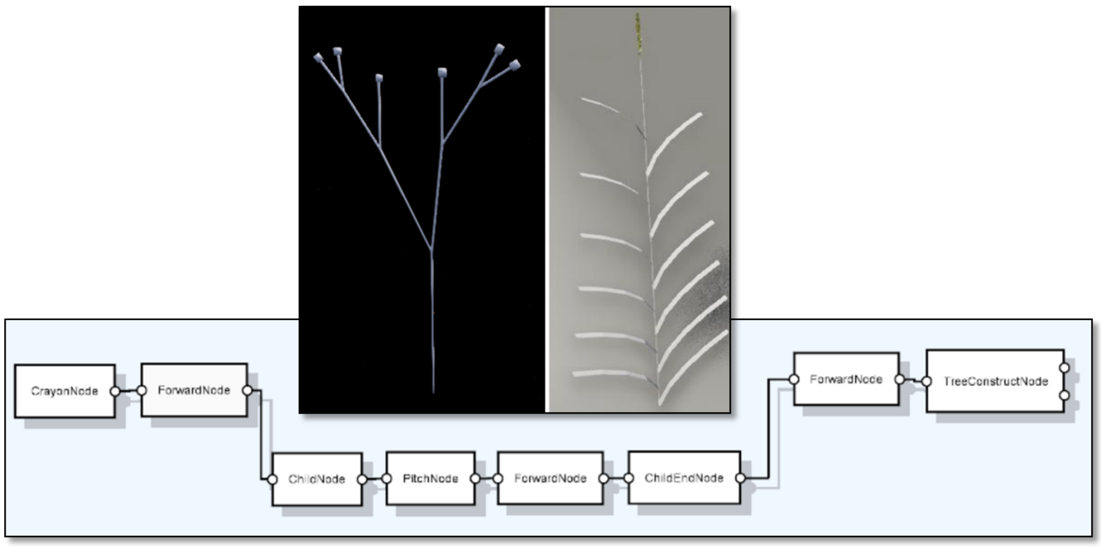


Figure 6.8: Example of a simple L-system rule and corresponding result in the scope of Benny Onrust's work approach within the Construct.

To generate plant models with L-systems, an external L-system library was employed. For it to be used within the Construct in a more a user-friendly way, most of the L-system symbols and commands were translated into Construct nodes. This is the case for common symbols such as Forward, Twist, and Pitch. By combining and connecting these symbols, users can define their own L-system rules.

In Figure 6.8, an example is shown of a simple L-system rule of this system with its output. The "Crayon" entity node at the beginning and the end indicates that the nodes in between are using the L-system library. All the basic L-system syntax is supported and integrated into the Construct. This way, the user is able to make its own L-system using a node-based approach.

6.7 Alfeite Base Reproduction

The Alfeite Base Reproduction project was a small project led by Ricardo Gonçalves, a student from the Faculty of Engineering of the University of Porto, around mid 2014. The final goal of this work [Gon14] was the expeditious creation of a 3D model of the coastal area in the Alfeite

Naval Base, Lisbon. This model's purpose was to be used in simulated training for search and rescue (SAR) missions, so as to better prepare the operators to the live tests that would occur in the real-world counterpart.

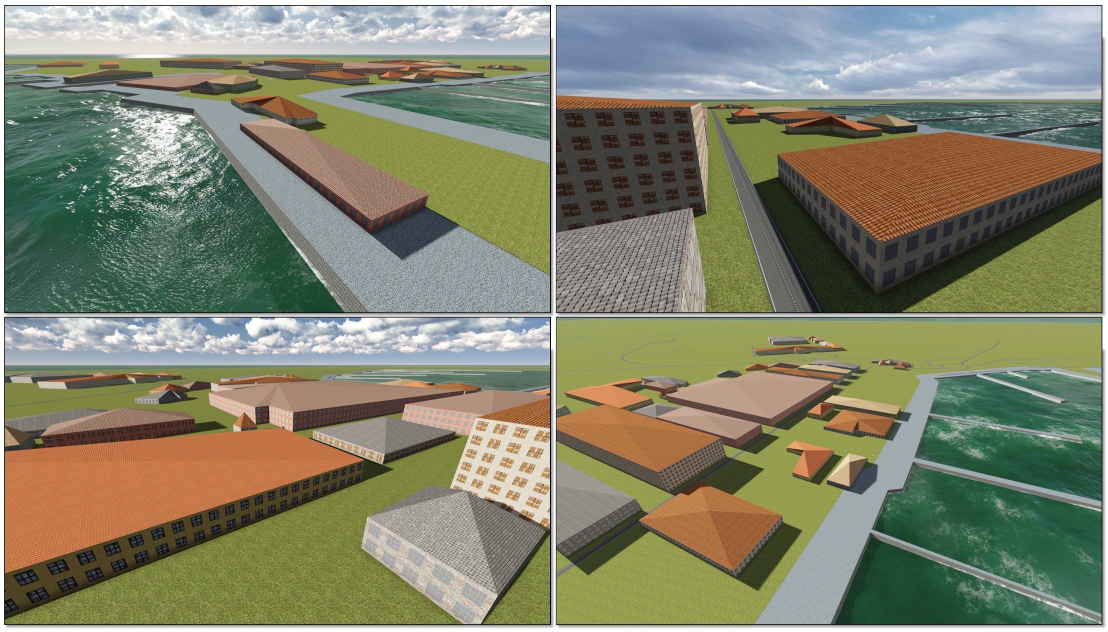


Figure 6.9: Alfeite Naval Base generated in the scope of Ricardo Gonçalves' work approach within the Construct [Gon14].

Given the scarcity of information of the intended area, one of the challenges of this work was the extraction of building altimetry. To address this, a solution that combined aerial imaging and geographical information to evaluate buildings shadows and calculate buildings heights was conceived. The building footprints, types and heights, along with street layout and coast profile information was then saved to an XML file, to be imported and processed within a Procedural Content Graph. A custom node was implemented to read the custom format and handled as standard Shape objects with associated attribute data, which was then fed to a sequence of geometry generation nodes, producing the final model shown in Figure 6.9.

6.8 Traffic Sign Generation

The Traffic Sign Generation project is an ongoing Master thesis project taken by Fieke Taal, a student from the Delft University of Technology, since mid 2014. At the moment of writing of this thesis, this project is still under development and therefore no concrete results are yet available.

The purpose of this Master is to study and develop procedural techniques for automatic placement of traffic signs in street and road networks (see Figure 6.10). This has required some research on existing regulations and standards in order to determine the rules that guide the assignment of certain traffic signs. Considered criteria for sign distribution should include the topology of the streets (e.g. crossings, tight curves), the neighboring buildings (e.g. schools,

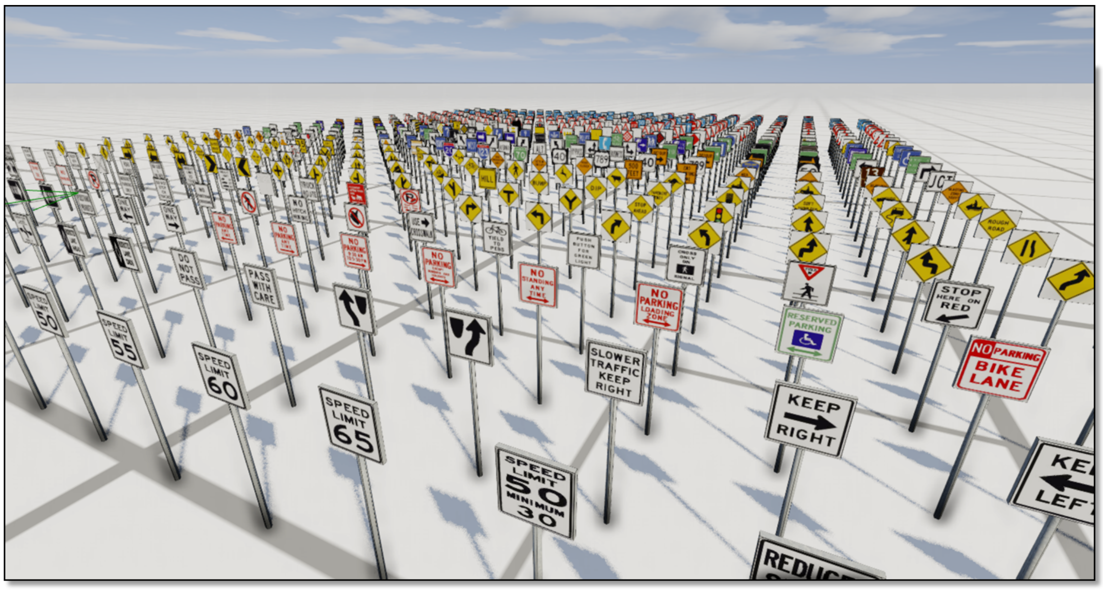


Figure 6.10: Traffic Signs to be generated within Fieke Tall's work approach.

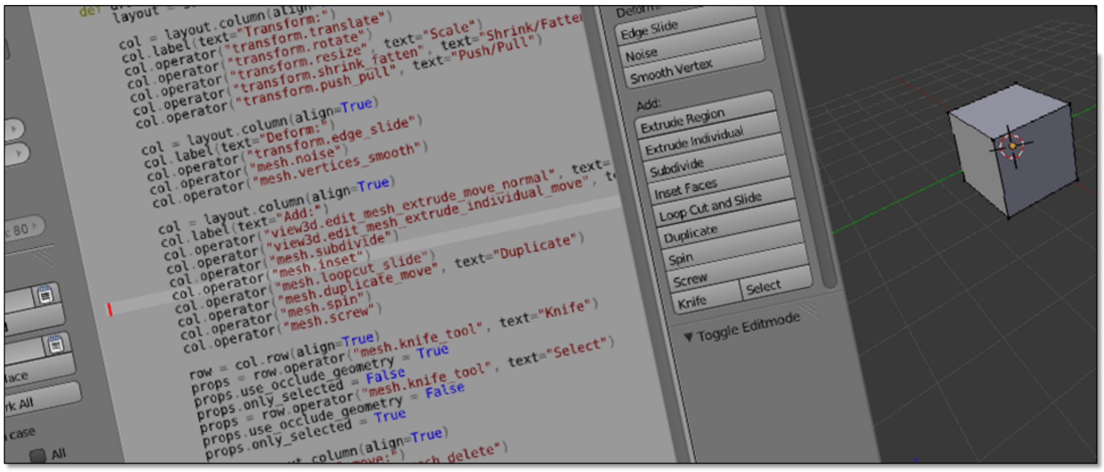
hospitals...), the environmental characteristics (e.g. terrain slope, type of area), location of points of interest (e.g. streets names, direction to cities), among other aspects that arise from a context-sensitive analysis. Currently researched solutions include graph traversal algorithms and spatial queries.

The ultimate goal of this project is to provide explicit traffic sign information and regulation in virtual urban environments, which can be used in entertainment or on more serious virtual reality applications, such as driving simulations.

6.9 Manual Modeling Integration

The Manual Modeling Integration project is a Master thesis project taken by Rui Gonçalves, a student from the Faculdade de Engenharia da Universidade do Porto, since late 2014. At the moment of writing of this thesis, this project is still at a starting point. The ultimate goal of this thesis is to provide the means to design and use procedural generation methods inside manual modeling tools.

The most common, manual modeling tools (Blender, 3Ds Max, Maya, etc.) offer a very fine control over vertex properties, edges and faces in an individual manner. These tools also feature the possibility to automate tasks and execute modeling algorithms through custom script writing - a work paradigm that typically is not attractive nor easy for the common user of such systems (see Figure 6.11). Procedural Content graphs arise as an alternative, but has only been implemented in its own environment. Hence, it is necessary to find ways to integrate such interaction paradigms, so as to benefit from the content creation facilities that each (manual and procedural) offers and, consequently, optimize the users' work process.



```
    layout
    col = layout.column(align
    col.label(text="Transform:")
    col.operator("transform.translate")
    col.operator("transform.rotate")
    col.operator("transform.resize", text="Scale")
    col.operator("transform.shrink_fatten", text="Shrink/Fatte
    col.operator("transform.push_pull", text="Push/Pull")

    col = layout.column(align=True)
    col.label(text="Deform:")
    col.operator("transform.edge_slide")
    col.operator("mesh.noise")
    col.operator("mesh.vertices_smooth")

    col = layout.column(align=True)
    col.label(text="Add:")
    col.operator("view3d.edit_mesh_extrude_move_normal", text=
    col.operator("view3d.edit_mesh_extrude_individual_move", t
    col.operator("mesh.subdivide")
    col.operator("mesh.inset")
    col.operator("mesh.loopcut_slide")
    col.operator("mesh.duplicate_move", text="Duplicate")
    col.operator("mesh.spin")
    col.operator("mesh.screw")

    row = col.row(align=True)
    props = row.operator("mesh.knife_tool", text="Knife")
    props.use_occlude_geometry = True
    props.only_selected = False
    props = row.operator("mesh.knife_tool", text="Select")
    props.use_occlude_geometry = False
    props.only_selected = True

          column(align=True)
```

Figure 6.11: A scripting window in Blender, which is not attractive for most users. [Wil12]

This project will require a study of the habits and needs of artists that use manual and procedural modeling tools. Only then can the best strategy be decided. A suggested solution consists in the development of an architecture, where Construct's procedure libraries can be integrated into such manual modeling tools as *modifiers*. These are typically low-level operations (noise, stretch, twist, uvw mapping, etc.), but could execute more complex, parameterizable procedures, such as the creation of buildings and complete cities, given a set of simple polygons. The final goal of this project is to facilitate the merging of the two methodologies, allowing users to develop procedural modeling processes that are integrated in a transparent way in the manual modeling system.

6.10 Mobile Procedural Urban Modeling

The Mobile Procedural Urban Modeling project is a Master thesis project taken by Vincent Hogendoorn since late 2014, a student from the Delft University of Technology. At the moment of writing of this thesis, this project is still at a starting point. The purpose of this project is to study the feasibility at using mobile devices as an input channel to procedurally model real-world buildings in a mixed-initiative approach.

The underlying motivation for this work is that, in order to reproduce real world structures with great fidelity, very detailed information is required. However, professional data acquisition is not only complex and expensive, but also prone to outdate due to the extensive and volatile nature of urban environments. In this sense, crowdsourcing appears as a solution for data collection from a set of contributors. Users, ideally the inhabitants, visitors or otherwise connoisseurs could provide knowledge of the urban area, so as to help the 3D virtual reconstruction of the urban space in question. Mobile devices are helpful tools for this purpose due to their features, including portability, location-awareness, photo capturing, web connectivity, etc. They would allow users to provide information directly when they are facing the real counterpart.

This requires some research on appropriate interaction methods, along with some investigation on how to harness the potential that such devices incorporate. For example, users could resort to the location-tracking features to select the building they would like to work on, use the camera to acquire material textures and use the compass/gyroscope to automatically determine the facade to which the material belongs. One of the many challenges lies on combining a high-level control of an evolving 3D parametric model with its procedural representation on a mobile device.

The intention is that the development of the intended methodology will profit from the usage of the Construct framework, whose set of libraries and API can be imported and used within mobile-compatible 3D engines, such as Unity or Monogame.

6.11 Summary

This chapter presented several case studies based on academic and research projects that have embraced the methodology proposed in this thesis, hence validating its usefulness and applicability.

As it can be verified by the number and variety of topics, Procedural Content Graphs are a versatile solution for generating several types of content using diverse strategies and operations. On the other hand, the flexibility and extensibility of the implemented solution has made the development of new features very easy to achieve, while its possibility for connection and incorporation within other systems has stimulated very unique concepts and ideas.

7. Discussion

In this chapter, the designed solution will be discussed, starting by comparing it with existing approaches, namely grammars and graph-based design tools. An analysis of the generation performance will come afterwards, where some obtained measurements will be listed and justified. Later on, some obtained feedback will be presented, followed by the results of a tutorial-based user study that was performed on the Construct.

7.1 Comparison with Existing Approaches

As mentioned in Section 2, grammar rules and other scripting-based approaches have become the most common means to control and manage procedural content generation processes, as they allow users to expressively define sequences of computational instructions that directly affect the evolution of a certain content [KK11; KPK10; LG06; M+06; Won+03; YK12]. Still, such approaches are generally unmanageable, unattractive and hard for artists to grasp, especially for those with little programming background. Some of the most common downsides of such textual approaches include [Sil+13]:

1. **Dataflow Visualization**: The fact that grammars are composed of rules makes the procedural process flow hard to visualize, as one must search matching rule symbols so as to understand the sequence of operations that are applied in order to generate a certain element.

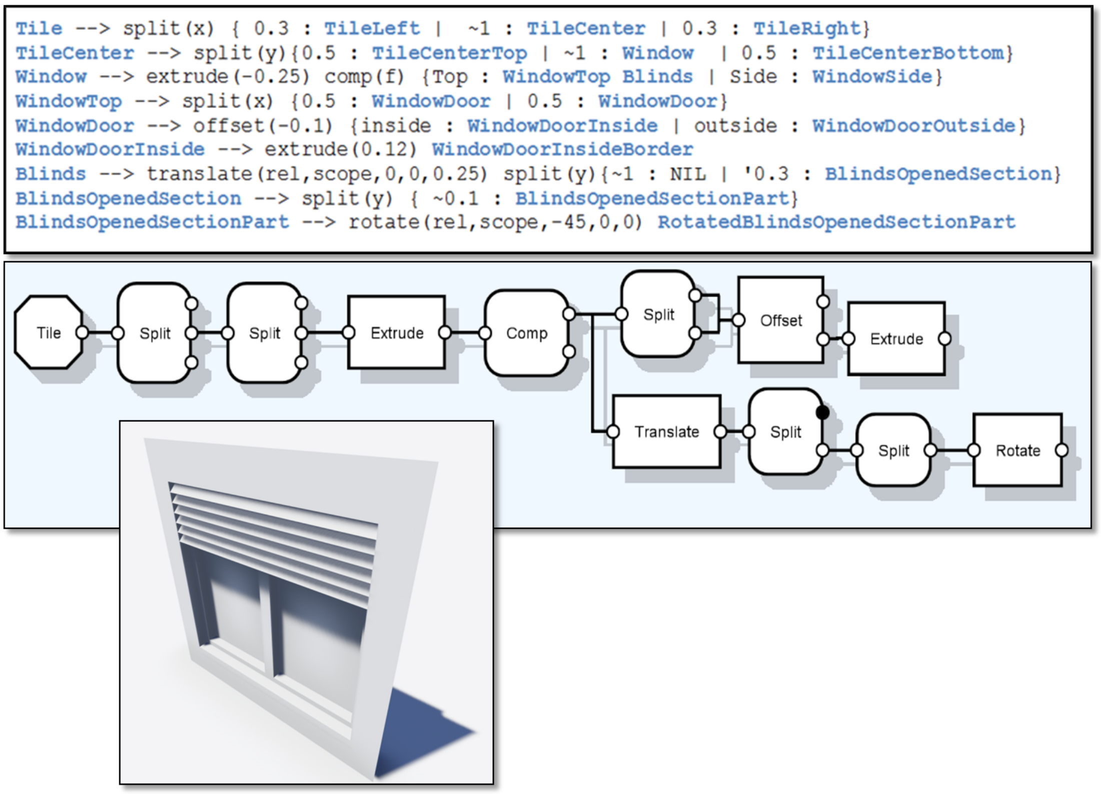
2. **Parameter passing**: In order to use auxiliary data to control the generation process, rules have to specify parameters and pass their values along to subsequent rules. If one has to pass a parameter from a starting rule to another that will only be necessary at the end of the chain, the process can become very cumbersome to write and manage.

3. **Rule naming**: The decomposition into rules requires each one to be identified with a symbol, which typically relates to the underlying operation or object being created. Still, not always can rules be named after something semantically relevant (see Figure 7.1), which makes them hard to understand by other users. Also, managing such symbols may lead to connectivity failures or erroneous rule connections, if one mistypes a symbol.

4. **Rule set encapsulation**: As any other programming paradigm, the possibility to reuse instruction sequences is a common feature. Still, not all grammar approaches support such a possibility, while those that do require the use of complex and verbose syntax declarations.

5. **Testing**: Testing and debugging a grammar are also challenging tasks, given their hard control over the sequence of operations scattered throughout the complicated rule chaining.

Procedural Content Graphs clearly address such grammar issues through their visual representation and management possibilities. Dataflow becomes clearer through the explicit representation of operations as nodes, while the source, direction and destination of the data flow is illustrated in the edges, always flowing from the left to the right, between ports of specific types. This also discards the need for rule names or labels, eliminating all issues related with symbol choice and management (see Figure 7.1).



```
Tile --> split(x) { 0.3 : TileLeft |  ~1 : TileCenter | 0.3 : TileRight}
TileCenter --> split(y){0.5 : TileCenterTop | ~1 : Window  | 0.5 : TileCenterBottom}
Window --> extrude(-0.25) comp(f) {Top : WindowTop Blinds | Side : WindowSide}
WindowTop --> split(x) {0.5 : WindowDoor | 0.5 : WindowDoor}
WindowDoor --> offset(-0.1) {inside : WindowDoorInside | outside : WindowDoorOutside}
WindowDoorInside --> extrude(0.12) WindowDoorInsideBorder
Blinds --> translate(rel,scope,0,0,0.25) split(y){~1 : NIL | '0.3 : BlindsOpenedSection}
BlindsOpenedSection --> split(y) { ~0.1 : BlindsOpenedSectionPart}
BlindsOpenedSectionPart --> rotate(rel,scope,-45,0,0) RotatedBlindsOpenedSectionPart
```

Figure 7.1: Window example modeled using CGA Shape grammars (loosely based on [M+06]) and Procedural Content Graphs. The visual nature of this approach makes the flow of data easier to follow and does not require the user to come up with rule names (highlighted) or geometric labels [Pat12; Sid15], which generally makes management more tiresome and error-prone.

In the same way, parameter passing becomes unnecessary, being replaced with the use of attributes, which, once defined, are accessible at any point of the graph. The reuse of node sequences is assured by the encapsulation possibilities that PCGRs incorporate, whereas its implementation has facilitated its use even more (see Section 5.2.2). Regarding testing, PCGRs already support several facilities for experimenting, verifying and debugging the generation process, which operate directly on the visual representation (see Section 5.2.4).

Regarding expressiveness, PCGRs have shown to be at least as powerful as grammars (see Section 4.1), being able to achieve the same manipulative power and design control. On the other hand, most grammars possess several limitations, which PCGRs attempt to address. In their top-down approaches, the splitting strategy of such grammars leads to independent parts, making linking and merging operations impossible. Although this has been somewhat addressed by passing *attachment points* along [KK11], being able to build bridge structures and cable connections, the fact is that no hints on performing aggregation, grouping and context-based operations were ever presented. This has been introduced by PCGRs through the support of collective and multi-port nodes, which are able to gather independent entities or combine them to perform operations based on relationships between entities.

Other graph-based approaches have also addressed such grammar limitations. The work of Patow [Pat12] is a general system with the goal of making shape grammars more comprehensive by allowing visualization in the form of a node-based system. Nonetheless, unlike PCGRs, the use of the Houdini engine [Sid15] makes it impossible to create cycles and therefore recursive procedures (a ground stone in grammars). In addition, this approach does not allow more than one output port, which could be used to control flow divergence. Instead, it resorts to label filtering, which introduces the same manageability issues as grammars for longer graphs, some of which have been alleviated by [BBP13]. Also, while it is possible to create high-level digital assets, it is not possible to introduce new types of semantic entities and their high-level nodes inherit the same (and other) port limitations.

For all the mentioned methodologies and tools, PCGRs appear to be the most comprehensive solution concerning data integration. In [KPK10], several non-terminal classes are provided, allowing the use of different geometry classes (e.g. FFD, Mesh, Polygon, Triangle), but, again, the methodology did not extend to other content domains, such as textures and application-specific objects. Likewise, visual node-based editors have become a standard for a variety of purposes and systems, including material editors (e.g. Autodesk Maya [Aut13a]), texture editors (Allegorithmic Substance Designer [All12]) script editors (e.g. Autodesk Softimage [Aut13b]) and model animation editors (SideFX Houdini [Sid15]), yet despite their advanced manipulative features, they lack the benefits of the integration of their methods in a single pipeline, an opportunity explored by PCGRs (see Section 4.3).

Regarding the implementation of the visual editor, the Construct should be considered a proof of concept and should not be compared directly to commercial tools such as CityEngine [Esr13] or Houdini [Sid15] when concentrating purely on design features (such as the sophistication of the available procedures) or interface accessibility. However, contrary to these systems, the Construct Framework has been designed with extensibility in mind, so that users can easily add their own components and manipulate their own custom entities.

7.2 Performance

The analysis of content generation performance is an important measure, as it indicates how much time the generation process requires to produce a certain result.

High generation performance is always preferred, although this may be more imperative in certain environments than others. An example is an game where the level has to be regenerated every time the game starts or where the objects have to be produced as the player flies throughout the environment (see section 6.1). Even when designing graphs in the Construct visual interface, since the constant re-execution of graphs is required, these should perform as quick as possible, so as to ensure a smooth experience to the user.

Table 7.1 contains a listing of the graph execution times for several of the examples presented in chapters 3 and 4:

Table 7.1: Generation times for several content complexities.

Model	Entity Count	Polygon Count	Vertex Count	Generation Time (ms)
Figure 3.3	1	275	608	35
Figure 3.5	53	313	836	81
Figure 4.1	374	107613	122944	7707
Figure 4.2	3782	46945	131599	37599
Figure 4.3	182	499	1048	321
Figure 4.4	4031	7564	22782	59873
Figure 4.7 a)	17099	118273	253444	4122
Figure 4.7 b)	2113	118273	253444	4122
Figure 4.10	2701	4001	13004	4698

The carried measures were obtained on an Intel® Core™ i7 - 2670QM, 2.2Ghz Laptop, featuring 16 GB RAM and an Nvidia® GeForce™ GTX 560M. Despite the large amount of available RAM, the resort to Microsoft XNA as a rendering engine has restricted the Construct visual interface to a 32-bit process, meaning that graph generation has not directly benefited from more than 4GB.

Performance varies per procedure, as it depends considerably on the type of processing algorithms, some of which are actually unfeasible or unavailable in other systems. It is important to note that the Construct features already a considerable number of procedures, and this focus on variety and control has left little time for optimizations. In fact, even simple optimizations could drastically reduce the values in Table 7.1.

Most important of all is the fact that the overhead time for node sequencing and data transport is negligible. This means that the graph representation and data flow execution algorithm itself does not constitute a performance issue.

For smaller and simpler cases, interactive graph modifications are possible and the user receives quick feedback. For larger examples, such as the one displayed in Figure 4.2, encapsulation and parametrization help reducing the generation scope to focus on smaller number of blocks or buildings at a time instead of on a vast area, therefore ensuring a smooth interactive experience. Several other optimization strategies can be employed to simplify certain graphs (see Section 4.2.9), to parallelize certain operations (see Section 3.8.5) or to avoid unnecessary generation repetitions (see Section 3.8.6), although all of them are dependent on the actual situation and type of content.

In addition to this measurements, no exhaustive performance comparison with other tools has been carried out, as it would go beyond the scope of the thesis. Construct, in its present state, should be considered as a proof of concept and therefore it is difficult to directly compare it to existing commercial tools. Still, generally speaking, it is possible to state that the execution times of PCGRs are not far from other tools, such as Houdini and CityEngine, for the same degree of geometric complexity.

Finally, one should note that, while performance is an important indicator, it is also naturally subject to improvement as hardware capabilities evolve. Considering the established goal for this thesis, it is the first priority to focus on an enhanced control, by the users, of the procedural content processes, even if such comes at a performance cost. Evidently, if one considers the necessary time to generate the same large-scale results using manual labor, then graphs certainly outperform manual approaches by many degrees of magnitude.

7.3 User Feedback

The gathering of user feedback was an activity that accompanied the conception process of Procedural Content Graphs and the implementation of the Construct. Being a methodology and consequently, a design tool to be employed by the common user, the concern has always been providing an attractive, flexible and intuitive alternative for the control of procedural generation processes.

However, it is important to underline that the focus of this thesis is not usability, but the empowerment of users with a methodology that allows them to express new design ideas and to integrate new types of content, all of which should be easily manageable. Achieving the latter lies essentially on the graph-based representation, which makes data flow clearer to understand, maintain and organize. That being said, the goal behind obtaining user feedback was to understand if such a new approach would be useful and advantageous when comparing to other ones, such as shape grammars or other graph-based systems.

Therefore, four users were asked to test the Construct and roughly ten others were simply presented their potential, most of them having had contact or actual experience with shape grammars (through CityEngine) or Houdini. Due to their experience with CGA Shape grammars and graph-based design tools, these participants were swiftly able to build basic structures and had first simple results within the first hour (see Figure 7.2).

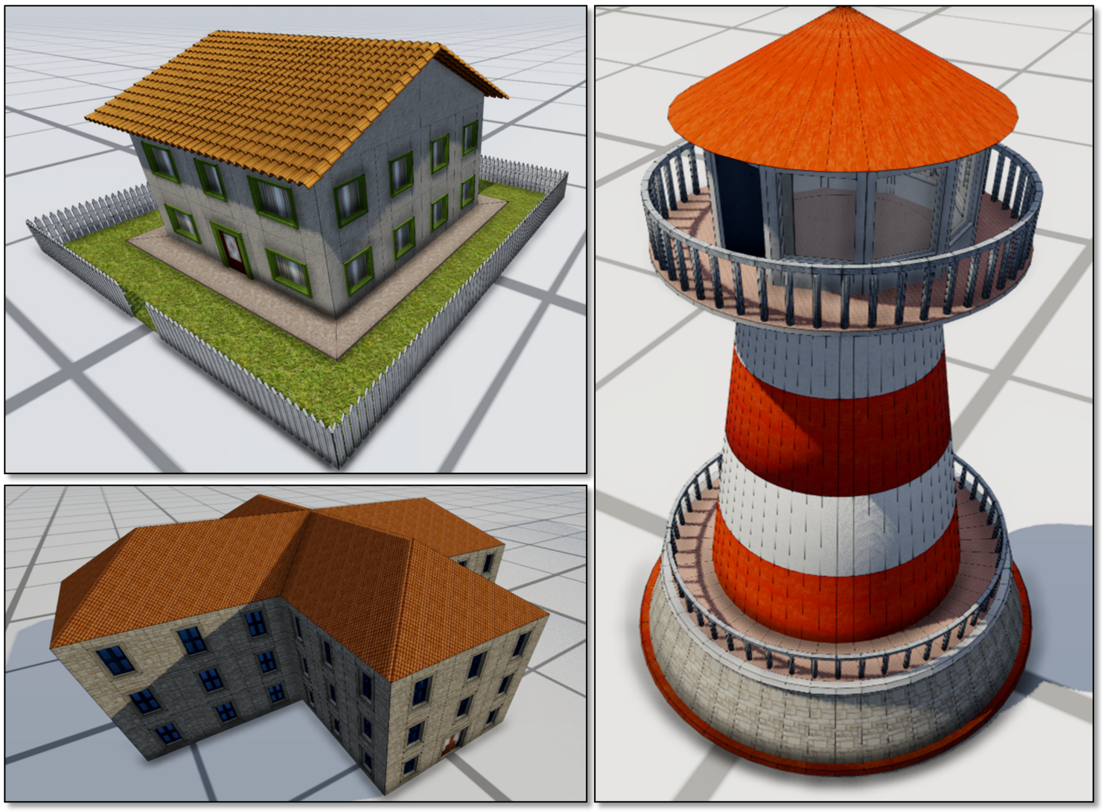


Figure 7.2: Examples of user-designed buildings using PCGRs.

All users were given only a few sample graphs to examine and had to figure out for themselves how the system worked. The responses led to the conclusion that the simple graph topology, node nomenclature and reduced graph size (achieved through successive encapsulation of meaningful operations) was a positive factor, which eased understanding, and was helpful for the following experimentation.

On a negative side, users commented on the difficulty to find out low-level nodes that they needed among the list of existing ones, leading to the reflection on the need for a proper categorization of the existing procedures. Examples that show control and merging of lists with geometric entities were most exciting to shape-grammar users. They understood and indicated that this new possibility would allow them to execute operations based on relationships between entities and they expected this feature to be very useful.

By now, the Construct system is already used by several research projects on procedural generation (see Chapter 6). Besides the ease-of-use, the researchers pointed out the great benefit of being able to easily define custom procedures and interactive tools.

7.4 Tutorial-based User Study

In late 2014, an opportunity arose to present the Construct and PCGRs to a group of students from the Delft University of Technology. These students belonged to a great diversity of study programmes, all around their 3rd Bachelor year, none of which from the computer science field.

Given the background of the students, the reduced available time for the experiment (2 hours) and the development status of the Construct at that moment of time, the experiment was simplified: the students were simply asked to follow a step-by-step tutorial (which can be consulted in Annex 1: Construct Tutorial). In the end, they were asked to answer a set of 8 simple questions on an online survey, 3 of which being presented as a Likert scale, so as to measure different degrees of opinion:

1. **Have you ever seen/used any graph-based system before?**
   If yes, can you name where/what software?

2. **How difficult did you find...**
   (In a 5-step scale from 1 - "Very Hard" to 5 - "Very Easy")
   (a) Interaction with the nodes and edges?

   (b) Understanding the data flow?

   (c) Manipulation of graph parameters and attributes?

   (d) Encapsulation of graphs into complex nodes?

   (e) Overall interaction with the Construct (project, inspector, 3D window...)?

3. **When building the graphs, how did you find...**
   (In a 5-step scale from 1 - "Not useful at all" to 5 - "Very Useful")
   (a) Expression builder?

   (b) Port coloring when hovered?

   (c) Graph encapsulation?

   (d) Edge disabling?

   (e) 3D Viewing options (view scope, edges...)?

4. **While using graphs, did you find anything particularly difficult or tiresome to do?**

5. **Did you find any tutorial sections particularly unclear, confusing and/or difficult? If so, which ones?**

6. **Considering the size of the model and the time that you took to build the neighborhood model, how would you rate...**
   (In a 5-step scale from 1 - "Very Bad" to 5 - "Very Good")
   (a) Quality of the produced models?

   (b) Model flexibility (how it can be tuned)?

   (c) User effort and efficacy?

   (d) Graph execution performance?

7. **Which other type of content would like to produce using the Construct?**

8. **What is your golden tip to improve the Construct?**

At the end of the experiment, 27 responses were obtained, with 22 being from male students and 5 from female students. Over 70% had never used or seen any graph-based system before, while the remaining ones mentioned having used tools such as Grasshoper for Rhino, Simulink, Blender and Unity.

As for the evaluation of graph and system interaction, the overall results were very positive (see Figure 7.3), with at most 20% of the inquired classifying as "Hard" one of the tasks. Interaction with the nodes and edges appeared to be accessible for all. The understanding of the data flow and the manipulation of graph parameters/attributes seemed to have posed some difficulties for some users, however the majority considered these tasks "Easy" or "Very Easy" to achieve. The overall interaction with the Construct appears to have faced a similar situation, but unfortunately the actual reason could not be investigated. Finally, the encapsulation appeared to be a challenge as well, with the majority of students having classified its execution difficulty as "Medium". This fact that could reside not only in the insufficient understanding of the practical, but also of the underlying theoretical concepts as well.

The third question focused on the evaluation of the usefulness of certain graph-development aids. The results, displayed in Figure 7.4, indicate that the expression builder, the port coloring and the graph encapsulation features were the most useful ones. A considerable fraction of the student sample, around 40%, seemed to be reasonably undecided regarding the usefulness of certain features, classifying them as "More or less useful", and others even worse. A possible reason is that most tutorial sections make little use of these features, provideding few expressive indications or opportunities to try with or without them. Such is particularly the case for edge disabling, which can explain why almost 30% of the inquired students has classified this interactive option as "Not very useful".

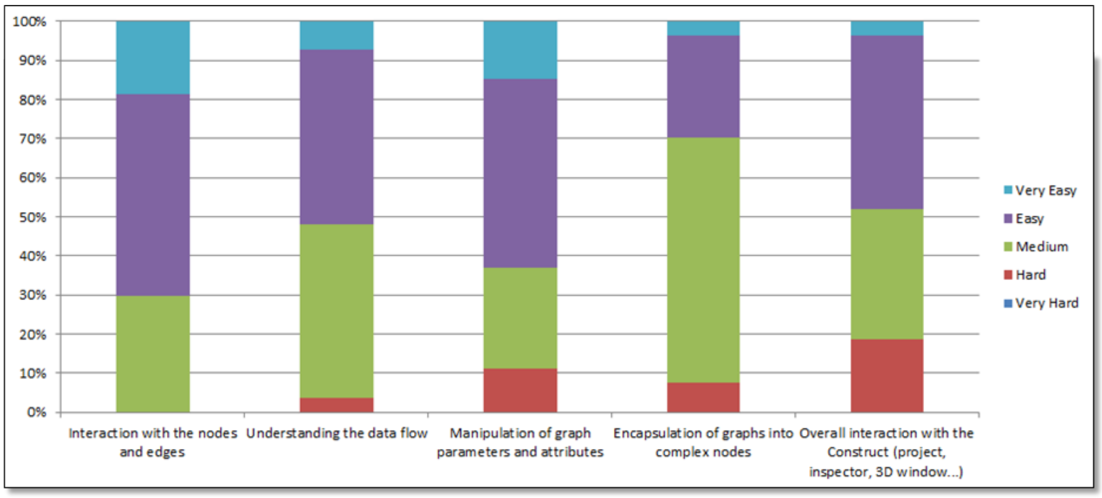


Figure 7.3: User classification of certain graph and system interaction tasks.

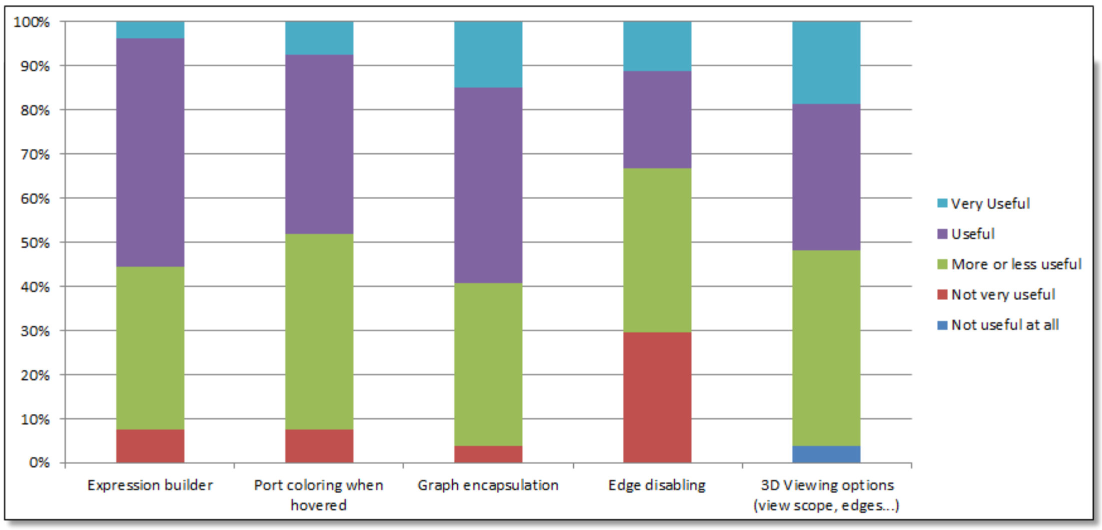


Figure 7.4: User classification of the usefulness of certain graph-development aids.

The fourth question was open answer and, while most opted not to respond or had nothing particular to note, few others returned some interesting and helpful tips. One of them suggested an automatic option to arrange all nodes in the canvas, as well as a quicker manner to configure edge connections, i.e. without having to first delete one to insert another. Some provided some small improvement tips to the procedure search window and pointed out that the expression editor syntax was evidently not intuitive or clear enough. Others made a comparison to Grasshopper, praising their smooth curve edges (which improves understanding and visibility) and their inspector window, which exists as an overlay on the canvas, instead of a side panel (making mouse interaction less tedious). Finally, many noted that the interface should include more keyboard shortcuts, which have a great impact on design usability and productivity.

The fifth question was again open for more detailed and varied answers. While most students were satisfied with the elaboration and completeness of the tutorial, some commented on a few difficulties in managing encapsulation and writing custom expressions, suggesting that a separate tutorial for expression building should be provided. Also, one revealed its insecurity to make himself something as complex after completing the tutorial, as the instructions were precise and required little self-experimentation and exploration. Aside from this, only some minor bugs and tutorial incorrections were noted.

The sixth question concerned a classification of the obtained result, considering the time and effort taken for the design of PCGRs (see Figure 7.5). Generally speaking, the results were again very positive, with at most 10% of the inquired feeling some discontent towards the user effort and efficacy, as well as the quality of the result. Graph execution performance was mostly considered "Good" or "Very Good", an opinion that was probably aided by the small scale of the modeled neighborhood and the organization of the tutorial, which focused on modeling small building sections at a time.

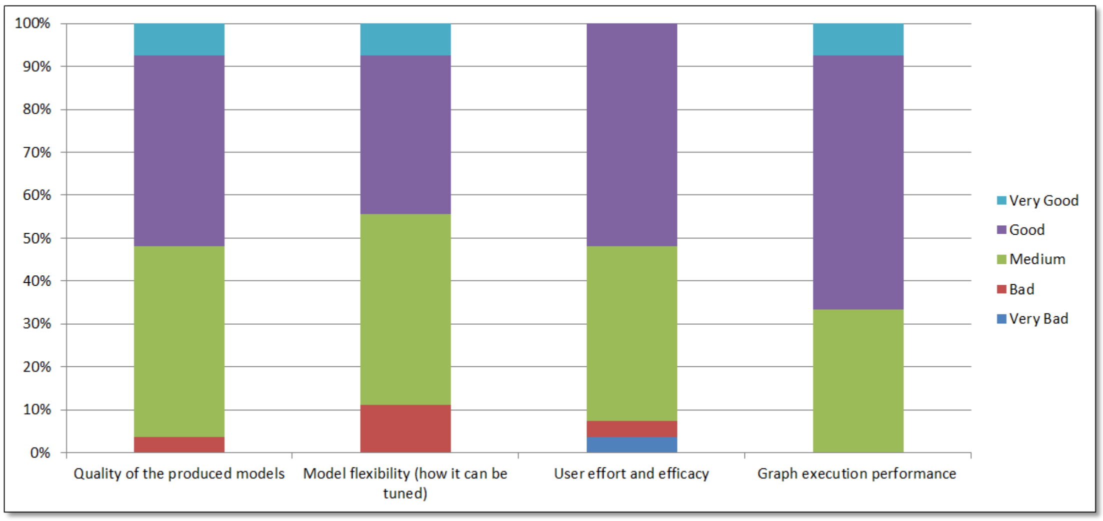


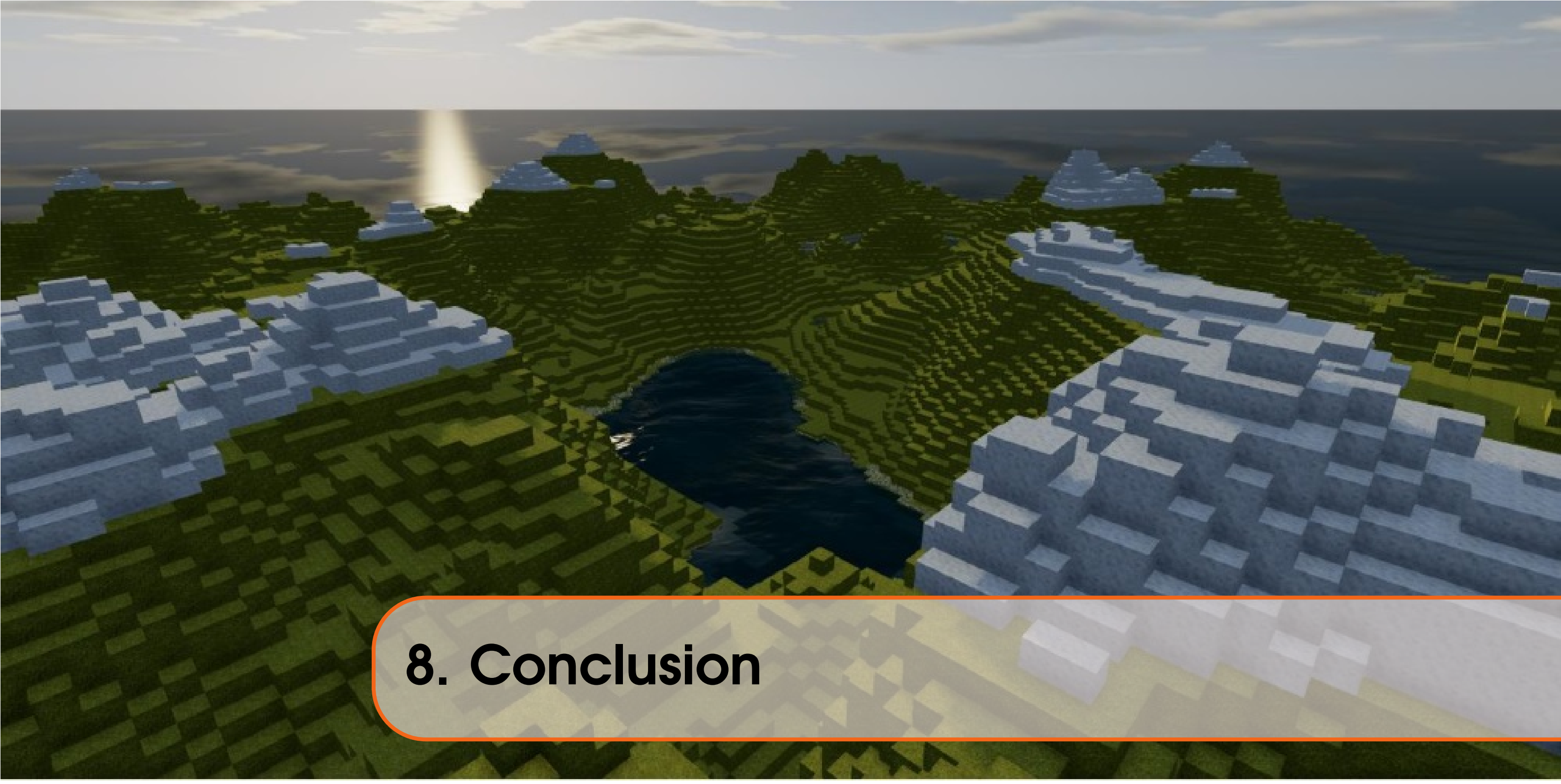
Figure 7.5: User classification of the obtained result.

The question about other suggested types of content received very diverse suggestions, including: natural landscapes in general, forests, complex buildings, caves, general props used in games (e.g. closets, cabinets, bins),cars, organic objects, behaviors. All of which constitute possible future types of content, but which will require the proper analysis and research about the best procedures to develop.

Finally, the last question reinforced the request for keyboard shortcuts, the implementation of undo/redo options in graph design and the addition of more advanced camera controls on the 3D viewer (e.g. speed adjustment, rotation around the object). Interestingly, one of the students suggested the idea of automatic graph generation based on direct interaction with the 3D model, a research possibility that is hard, but not impossible to achieve with the current Construct implementation.

7.5 Summary

The results of both the methodology and implementation were discussed in this chapter. Having the actual design possibilities already been presented in the fourth chapter, a comparison to other methodologies and tools ensued, where the various benefits of this new approach were enumerated and explained. Generation performance was also subject of discussion, based on some acquired metrics, and, while considered in par with other alternatives, several optimizations can still be achieved. When letting users experiment on the system, they were able to develop their own graphs and produce interesting results. In general, the feedback was very positive, yet the testing was also very useful for gathering improvement suggestions.



8. Conclusion

This Ph.D. thesis document, developed in the scope of the Doctoral Program in Informatics Engineering, on the field of Computer Graphics and Gaming, consisted in the exposition of the concept, implementation and experimentation of Procedural Content Graphs, a new visual programming language aimed at improving the design expressiveness, manageability and integration in procedural content generation.

This chapter will present a summary of the main research contributions of this approach, answering the raised research questions, after which several recommendations, ideas and improvements considered for future work will be enunciated.

8.1 Objective Completion

At this final phase of this Ph.D. thesis, it would be accurate to state that all the initial proposed objectives were accomplished, with quite satisfactory results. As such, it is possible to answer the raised research questions:

***1.** How can expressiveness be improved to achieve a more broad and comprehensive range of designs?*

One of the major goals behind Procedural Content Graphs was the creation of a new language that would allow the expression of new concepts, ideas and operations, as well as the production of unaddressed designs and structures, surpassing limitations present in grammars or other existing rule-based approaches.

This has been made possible through the generic, modular and extensible representation thoroughly explained in chapter 3. It is based on nodes that can perform any imaginable operation over both individual and collections of entities, and therefore express design intentions that affect content on both local and global scale. Through the use of aspects and augmentations, different types of techniques and strategies can be configured and employed, ranging from more imperative instructions to more declarative ones, where it is possible to rather specify what to achieve, instead of how. This is also reinforced through the resort to encapsulation, allowing for an operation at various levels of abstraction, i.e. so that one can opt to specify details using low-level operations or rapidly combine higher-level architectural blocks.

This expressive power is demonstrated throughout Chapter 4 through various examples of constructions that could not be achieved using previous approaches, especially due to their lack of control over lists of entities, which would limit data manipulation abilities and therefore restrict the application of certain methodologies. Aggregation, clustering, combination and ray casting are particular examples of operations that allow for contextual development, which ultimately facilitates the specification of certain patterns and relationships.

Finally, this expressiveness is also evidenced in its materialization as the Construct Framework, described in Chapter 5, which features the possibility to implement new procedures, aspects and entities, so that new problems can be addressed using new approaches that maximize design efficacy.

***2.** How should procedures and objects be represented, so as to benefit from their inherent semantics and improve content manageability?*

The visual representation of Procedural Content Graphs is a major benefit that makes content flow not only easier to specify, but also to maintain and operate on. When compared to grammar specifications, whose textual-based nature makes the process unattractive and hard to manage, the use of nodes to define transformation procedures and the use of edges to build connections between them makes the flow of data more clear and easy to follow, a fact discussed and verified through user experimentation in Section 7. Also, the use of encapsulation contributes to a better separation of concerns, the reusability of large operation chains and the organization of nodes into more semantically-rich building blocks. The introduction of dynamic loading is another feature that can help in the encoding of the generation process, by using surrogate nodes to execute procedures under specific conditions, a specific number of times, with particular variations.

Manageability also refers to the facility to manipulate and combine entity data in a meaningful way to achieve a certain result. On the one hand, this is aided by the typed nature of entities, which can feature more concrete structures to answer a particular need (e.g. terrain, streets) and be recursively composed so as to build a semantic hierarchy for better organization, understanding and handling of data between operations. The power to create graph attributes is also fundamental to provide a flexible definition of entity semantics and better control the flow of content, especially when used in conjunction with encapsulation. On the other hand, the constitution of nodes themselves and their support for multiple ports make both combination and separation of data flow possible. Together with the introduction of gather ports, the manipulation of data as lists has arisen, enlarging the range of possible operations on the content, such as aggregation, merging and context-based verifications. Finally, the representation of entities as objects that can be handled through their references introduced new alternatives for data manipulation, making management practices more flexible and the control over entity hierarchies more graceful to achieve.

***3.** How can different types of content be generated and integrated within a single pipeline and what advantages does it encompass?*

Procedural Content Graphs are naturally capable of manipulating different types of content, as long as they are represented as subtypes of the entity type and can be subjected so the same type of control and flexibility. The ability to manipulate a certain type lies on the existence of procedures that are capable of generating or transforming them, a fact typically reflected on the typing of the node input and output ports, explained in Chapter 3.

When considering the extensive nature of urban environments, developing and putting geometry, textures, lights and behaviors together on a separate manual step can become a tiresome task, as well as lead to gaps and inconsistencies in the final result. Handling different types of content within the same pipeline has foremost the advantage that a more comprehensive result can be produced in one go. Another advantage lies in the fact that the different types of entities interact and influence one another. Such is the case of the geometry scope that determines the location of the door and the corresponding trigger button, or the example of street that affects the texture to be applied on the sign indicating the street name.

Both the possibilities of content definition and integration are present in the implementation of the Construct Framework, which promotes connection and incorporation within other systems and, consequently, the reception and communication of external data that affects the generation process, for exchanging command, analysis and report information or to export the produced content either directly or through intermediate formats to other systems.

In summary, considering the designed methodology around procedural content graphs and its implementation with the Construct, which provides greater expressive power, manageability and integration possibilities to control procedural content generation algorithms and, consequently, to produce more complete virtual urban environments, the stated hypothesis can be verified:

The production of comprehensive virtual urban environments, integrating diverse types of content, can be better expressed and managed by a unified rule specification language to guide procedural content generation methodologies.

8.2 Future work

Throughout the development of this thesis, as the methodology grew more mature and the Construct implementation more complete, several research lines have arisen, having been projected and outlined but, due to time limitations, they could not be addressed during this thesis. On the other hand, PCGRS, however expressive and manageable as they can be, can still be further studied, improved, and complemented in several ways. Same applied to the development of the Construct, which was especially designed with extensibility in mind. Having all this in mind, future work will focus on the following:

Usability Testing and Improvement: In addition to the acquired user feedback, no formal user study was conducted to quantitatively investigate user-friendliness, as it would go beyond the scope of the thesis. The reasoning was that, given the prototypical nature of the Construct, many usability features (such as keyboard shortcuts, visual aids, etc.) would require a considerable amount of time to implement or fix, despite not concerning to PCGRs in particular. Therefore, one task will be to address such features, performing the said usability studies subsequently.

Node Categorization and Polishing: During the development of the various procedure libraries mentioned in Section 5.1, several nodes were implemented. While enhancing the manipulation possibilities of several content types, it also contributed to a more complex management and understanding of each node's functionality and application. Also, the fact that new procedures and entities can be easily programmed and introduced to the system leads often to the creation of basic nodes that could also be represented using simple combinations of other nodes. Future efforts in this sense will consist in finding ways to categorize and merge certain operations, as well as to analyze what basic operations are indeed required, so that an easier management of the existing nodes can be guaranteed.

Generation of New Content Types: Given the ease of introduction of new entity types and procedures, future work will focus on the development of further content types and structures. Aside from the decision on the representation of the content and the means to visualize and test it, a considerable effort will involve a some study on the necessary procedures to handle it, or on the adaptation of existing methodologies to the graph-based paradigm. Examples of new content types are characters, vehicles, weapons, household materials (e.g. closets, cabinets, pots, bins), etc.

Interactive Graph Generation: Despite the facilities and popularity of graph-based approaches, the preferable paradigm of content design still consists in directly manipulating the final result interactively. For instance, in 3D modeling, this is achieved through picking possibilities and subsequent transformations, while in images one can modify custom sections and layers of pixels using the mouse or other peripherals. In procedural generation, achieving the same type of interaction is more difficult, because it focuses rather on the specification of rules, not on simple changes on a static resource. A future work goal will therefore consist in attempting to generate PCGRs automatically from user interaction on a resource, such as a 3D model. This is quite a challenge, as the design intent is often ambiguous ([LWW08; Sme+11]). For instance, the selection of a face can have several possible interpretations, such as being the one having particular characteristics, e.g. an index, a color or facing direction. Or, the translation of a vertex may be the result of a calculation involving properties of related entities, a concept even harder to interpret. Still, even if a generic solution cannot be achieved, displaying a range of possibilities and completing the graphs automatically for simpler cases can already ease the effort of the user considerably.

Interactive Configuration: A simpler alternative to graph generation consists of content-interactive handles that, once manipulated, simply affect the parametrization of certain nodes. This can be achieved by letting procedures themselves produce such handles at the same time as they produce a certain resource. For instance, a translation operation should return three perpendicular arrows that, once pulled, will modify the X, Y or Z parameter values of the corresponding node. Similar ideas have been presented in [Esr13; KK12], yet were limited to the application of geometric data and could only be applied to a final model, while having no strategy to propagate such handles throughout encapsulation. Such restrictions would be addressable on PCGRs.

Research Tool Integration: There are many approaches and systems developed in the scope of other academic and research projects which have proven to be very effective in the generation of particular content types through specific interaction means. A possible endeavor would be to integrate such systems in the Construct Editor, or integrate the Construct Framework in those systems. One example is the software SketchaWorld [Sme+11]: its declarative and interactive approach would complement PCGRs, but also benefit from its expressive power to define procedural generation processes.

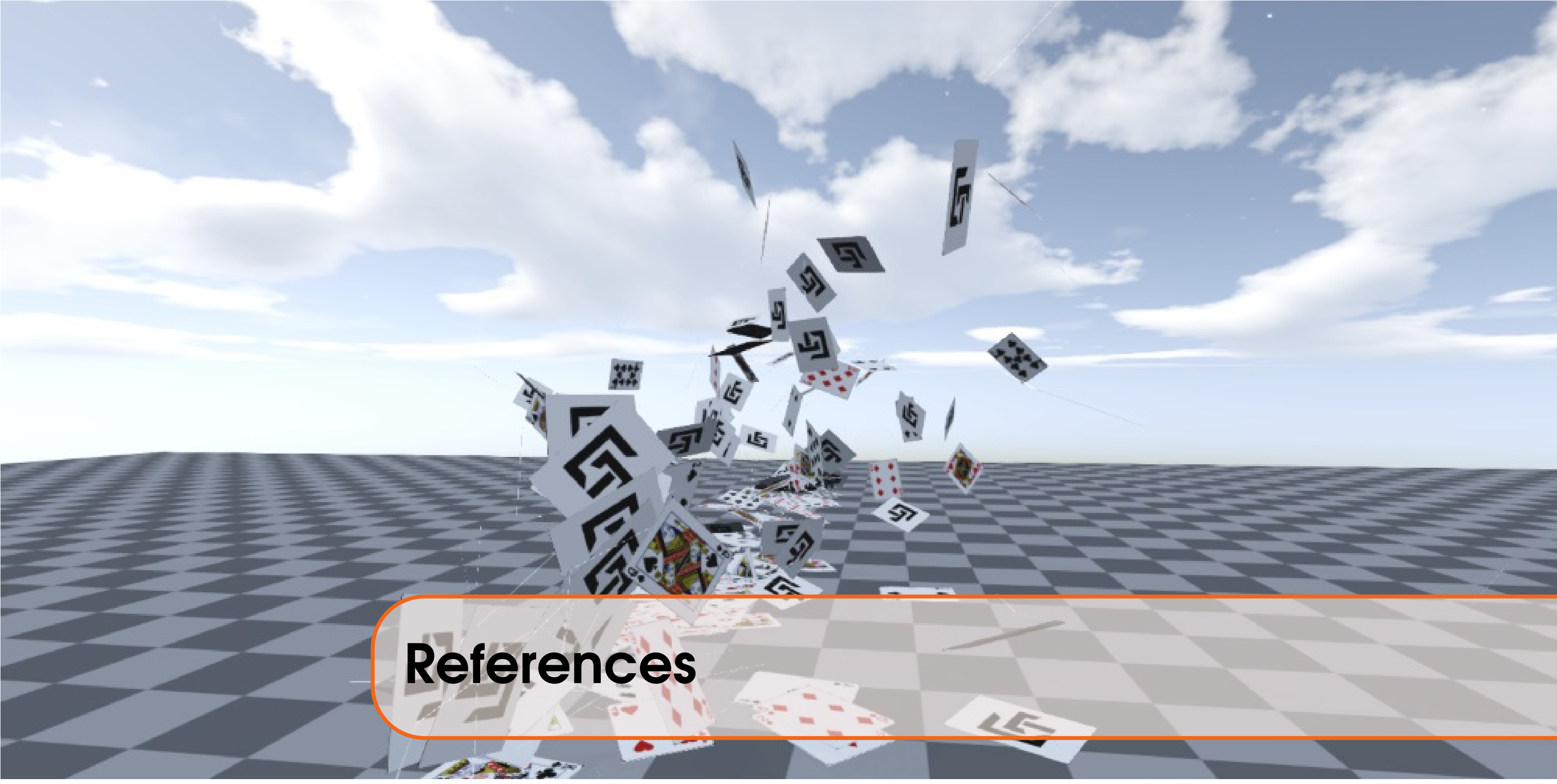
More Constraint and Context-based Operations: The possibilities to employ constraint-based and context-based operations have been demonstrated in Chapter 4, yet in practice very few of such operations have been explored. This was caused essentially by their implementation complexity, as well as the need for closer consideration on how they should be designed and how they should operate. The development of such operations will be subject of more attention in the future.

Meta-procedural Generation: By this concept is meant the procedural generation of PCGRs themselves, which in turn generate the actual content. This idea, first explored after the internship with the Esri Research & Development Center in Zurich, consists in performing automatic recombination of graph sets. The goal is to generate variations of existing graph designs with little additional effort, by letting an algorithm analyze established connections between certain nodes and replicating them with some randomness. Evidently, to avoid chaotic combinations and the production of arbitrary results, this process will ideally be applied over higher-level encapsulated nodes so as to guarantee meaningful combinations of elements such as windows, balconies, pillars and other architectural elements.

Optimization and Profiling: As mentioned in Section 7.2, despite being on par with other tools, the generation performance of the Construct implementation can still undergo some optimizations. Doing so will first require the development of better profiling tools, so as to more accurately detect the source of inefficiencies and unnecessary overheads. Only then can certain procedures be improved, reducing the necessary time for generation. Alternatively, this may also involve changing generation strategies, by adopting completely new ones that produce similar results using faster method combinations.

More Application within Games and Simulations: Although it has been proven that both the methodology and implementation are powerful enough to develop comprehensive environments, including of application-specific content, a fact is that, until this point, there have been very few opportunities to employ such resources in practice. Therefore, future enterprises will certainly involve some closer work with game development teams or companies, so as to study and understand the interest and applicability of such generated content in actual games and simulations.

Community-driven development: One of the main concerns when implementing the Construct system was the development of an API that would be easy, powerful, generic and attractive to use. Having achieved that goal, the next step will be to make it accessible to others so as to raise an active community that can contribute with new features. This will require proper disclosure, the conception of supporting documentation and help guides, as well as the creation of a platforms where users can consult and discuss ideas, exchange designed graphs and share libraries of implemented procedures. This way, one can expect more advanced procedural generation methodologies and tools to be developed in the future.



References

[Aco88] Acornsoft. *Elite*. 1988 (cited on page 40).

[Aa+14] Telmo Adão, Luís Magalhães, Emanuel Peres, and Francisco Pereira. "Procedural Generation of Traversable Buildings Outlined by Arbitrary Convex Shapes". In: *Procedia Technology* 16 (2014), pages 310–321. ISSN: 22120173. DOI: 10.1016/j.protcy.2014.10.097. URL: http://www.sciencedirect.com/science/article/pii/S2212017314003247 (cited on page 136).

[Ale+05] Al Alexander, Tad Brunyé, Jason Sidman, and Shawn Weil. "From gaming to training: A review of studies on fidelity, immersion, presence, and buy-in and their effects on transfer in pc-based simulations and games". In: *DARWARS Training Impact Group* November (2005), page 14. ISSN: 15526259. DOI: 10.1016/j.athoracsur.2004.02.012 (cited on pages 30, 31).

[Ali+09] Saif Ali, Jieping Ye, Anshuman Razdan, and Peter Wonka. "Compressed facade displacement maps." In: *IEEE transactions on visualization and computer graphics* 15.2 (2009), pages 262–73. ISSN: 1077-2626. DOI: 10.1109/TVCG.2008.98. URL: http://www.ncbi.nlm.nih.gov/pubmed/19147890 (cited on page 104).

[All12] Allegorithmic. *Allegorithmic Substance Designer*. 2012. URL: http://www.allegorithmic.com/products/substance-designer (cited on page 149).

[Aut13a] Autodesk Inc. *Autodesk Maya*. 2013. URL: http://usa.autodesk.com/maya/ (cited on page 149).

[Aut13b] Autodesk Inc. *Autodesk Softimage*. 2013. URL: http://autodesk.com/softimage (cited on page 149).

[Bar00] Michael Barnsley. *Fractals everywhere*. Morgan Kaufmann Pub, 2000. ISBN: 0120790696 (cited on page 46).

[BBP13] Santiago Barroso, Gonzalo Besuievsky, and Gustavo Patow. “Visual copy & paste for procedurally modeled buildings by ruleset rewriting”. In: *Computers and Graphics (Pergamon)* 37 (2013), pages 238–246. ISSN: 00978493. DOI: 10.1016/j.cag.2013.01.003 (cited on pages 55, 149).

[BA05] Farès Belhadj and Pierre Audibert. “Modeling landscapes with ridges and rivers: bottom up approach”. In: *Proceedings of the 3rd international conference ...* Volume 1. ACM, 2005, pages 1–4. ISBN: 1595932011. DOI: 10.1145/1101389.1101479. URL: http://dl.acm.org/citation.cfm?id=1101479 (cited on page 46).

[Ben09] Benzin. *Procedural Generation in Infinity*. 2009. URL: http://forgottenportal.com/infinityblog/2009/01/procedural-generation/ (cited on page 41).

[Bet14] Bethesda Softworks LLC. *The Elder Scrolls Official Site*. 2014. URL: http://www.elderscrolls.com/ (cited on page 40).

[Bit02] Barry Bitters. “Feature Classification System and Associated 3-Dimensional Model Libraries for Synthetic Environments of the Future”. In: *Proceedings of I/ITSEC 2002 Conference*. 2002 (cited on page 34).

[Bli12] Blizzard Entertainment Inc. *Diablo 2*. 2012. URL: http://us.blizzard.com/en-us/games/d2/ (cited on page 40).

[Bor08] Borderlands Guide. *How Do You Make Over Half A Million Guns?* 2008. URL: http://borderlandsguide.com/news/how-do-you-make-over-half-million-guns (cited on page 40).

[BN08] Eric Bruneton and Fabrice Neyret. “Real-time rendering and editing of vector-based terrains”. In: *Computer Graphics Forum* 27.Eurographics 2008 (2008), pages 311–320 (cited on pages 42, 48, 49).

[Che+08] Guoning Chen, Gregory Esch, Peter Wonka, Pascal Müller, and Eugene Zhang. *Interactive procedural street modeling*. Los Angeles, California, 2008. DOI: 10.1145/1399504.1360702 (cited on pages 49, 50).

[Cho56] Noam Chomsky. “Three models for the description of language”. In: *IRE Transactions on Information Theory* 2.3 (1956), pages 113–124. ISSN: 0096-1000. DOI: 10.1109/TIT.1956.1056813 (cited on page 53).

[Coe+07] António Coelho, Maximino Bessa, António Augusto Sousa, and Fernando Nunes Ferreira. “Expeditious modelling of virtual Urban environments with geospatial L-systems”. In: *Computer Graphics Forum* 26.4 (2007), pages 769–782. ISSN:

01677055. DOI: 10.1111/j.1467-8659.2007.01032.x. URL: http://dx.doi.org/10.1111/j.1467-8659.2007.01032.x (cited on pages 42, 58–60, 62, 63, 102).

[Dan07] Danc. *Lost Garden: "Content is Bad"*. 2007. URL: http://lostgarden.com/2007/02/content-is-bad.html (cited on pages 37, 38).

[Deu+98] Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomir Měch, Matt Pharr, and Przemyslaw Prusinkiewicz. "Realistic Modeling and Rendering of Plant Ecosystems". In: *Conference on Computer Graphics and Interactive Techniques*. ACM, 1998, pages 275–286. ISBN: 0897919998. DOI: 10.1145/280814.280898. URL: http://portal.acm.org/citation.cfm?id=280814.280898 (cited on page 47).

[Dis09] Ben Discoe. *Artificial Terrain Generation*. 2009. URL: http://www.vterrain.org/Elevation/Artificial/ (cited on page 43).

[DB05] Jürgen Döllner and Henrik Buchholz. "Continuous level-of-detail modeling of buildings in 3D city models". In: *Proceedings of the 13th annual ACM international workshop on Geographic information systems* (2005), pages 173–181. DOI: 10.1145/1097064.1097089. URL: DOLN05http://portal.acm.org/citation.cfm?doid=1097064.1097089 (cited on page 58).

[Dou08] Andrew Doull. *The Death of the Level Designer: Procedural Content Generation in Games*. 2008. URL: http://roguelikedeveloper.blogspot.com/2008/01/death-of-level-designer-procedural.html (cited on pages 36, 40).

[Esr13] Esri. *Esri CityEngine | 3D Modelling Software for Urban Environments*. 2013. URL: http://www.esri.com/software/cityengine/ (visited on 02/21/2013) (cited on pages 49, 53, 58, 96, 149, 163).

[FS05] John Feil and Marc Scattergood. *Beginning Game Level Design*. Thomson Course Technology, 2005, page 252. ISBN: 1592004342. URL: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&\#38;path=ASIN/1592004342 (cited on page 31).

[Fin08] Dieter Finkenzeller. "Detailed Building Facades". In: *Computer Graphics and Applications, IEEE* (2008), pages 58–66. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4497898 (cited on pages 41, 62).

[FFC82] Alain Fournier, Don Fussell, and Loren Carpenter. "Computer rendering of stochastic models". In: *Communications of the ACM* 25.6 (1982), pages 371–384. ISSN: 00010782. DOI: 10.1145/358523.358553 (cited on page 43).

[GMSe09] James Gain, Patrick Marais, and Wolfgang Straß er. "Terrain sketching". In: *Proceedings of the 2009 symposium on Interactive 3D graphics and games - I3D '09*. ACM, 2009, page 31. ISBN: 9781605584294. DOI: 10.1145/1507149.1507155. URL: http://dl.acm.org/citation.cfm?id=1507149.1507155 (cited on page 44).

[Gal+10] Eric Galin, Adrien Peytavie, Nicolas Maréchal, and Eric Guérin. "Procedural generation of roads". In: *Computer Graphics Forum*. Volume 29. 2. Wiley Online Library, 2010, pages 429–438. ISBN: 1467-8659. DOI: 10.1111/j.1467-8659.2009.01612.x (cited on page 51).

[Gal+11] Eric Galin, Adrien Peytavie, Eric Guérin, and Bedřich Beneš. "Authoring Hierarchical Road Networks". In: *Computer Graphics Forum* 30.7 (Sept. 2011), pages 2021–2030. ISSN: 01677055. DOI: 10.1111/j.1467-8659.2011.02055.x. URL: http://doi.wiley.com/10.1111/j.1467-8659.2011.02055.x (cited on page 67).

[Gal11] Alex Galuzin. *Ultimate Level Design Guide*. 2011. URL: UltimateLevelDesignGuide (cited on page 31).

[Gon14] Ricardo Gonçalves. *Project Alfeite – The 3D Reconstruction of the Alfeite Naval Base*. Technical report. 2014, page 12 (cited on pages 141, 142).

[Goo10] Google. *Google Earth*. 2010. URL: http://earth.google.com/ (cited on page 30).

[Gre+03] Stefan Greuter, Jeremy Parker, Nigel Stewart, and Geoff Leach. "Real-time procedural generation of 'pseudo infinite' cities". In: *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*. GRAPHITE '03. Citeseer, 2003, 87–ff. ISBN: 1581135785. DOI: 10.1145/604487.604490. URL: http://portal.acm.org/citation.cfm?doid=604471.604490 (cited on pages 58, 59).

[Gu6] B Guðlaugsson. "Procedural Content Generation". In: (2006) (cited on pages 37, 38).

[Hal08] Luke Halliwell. *Procedural content generation*. 2008. URL: http://lukehalliwell.wordpress.com/2008/08/05/procedural-content-generation/ (cited on pages 36, 37).

[Ham01] Johan Hammes. "Modeling of Ecosystems as a Data Source for Real-Time Terrain Rendering". In: *Framework* (2001), pages 98–111. DOI: 10.1007/3-540-44818-7_14 (cited on page 47).

[Han11] Hanbitsoft Inc. *Hellgate: London*. 2011. URL: http://hellgate.t3fun.com/ (cited on page 40).

[Har09] Harvard University. *Digital Elevation Models*. 2009. URL: http://www.gsd.harvard.edu/gis/manual/dem/ (cited on page 45).

[Hna+10] Houssam Hnaidi, Éric Guérin, Samir Akkouche, Adrien Peytavie, and Éric Galin. "Feature based terrain generation using diffusion equation". In: *Pacific Graphics*. Volume 29. 7. Wiley Online Library, 2010, pages 2179–2186. ISBN: 1467-8659 (cited on pages 43, 45).

[HYN03] Jinhui Hu, Suya You, and Ulrich Neumann. "Approaches to Large-Scale Urban Modeling". In: *IEEE Comput. Graph. Appl.* 23.6 (2003), pages 62–69. DOI: http://dx.doi.org/10.1109/MCG.2003.1242383 (cited on page 42).

[InDM06] J Ibáñez and C Delgado-Mata. "A Basic Semantic Common Level for Virtual Environments." In: *IJVR* 5.3 (2006), pages 25–32. URL: http://scholar.google.com/scholar?hl=en\&btnG=Search\&q=intitle:A+Basic+Semantic+Common+Level+for+Virtual+Environments\#0 (cited on page 34).

[IDV11] IDV Inc. *Speedtree*. 2011. URL: http://www.speedtree.com/ (cited on page 48).

[Int07] Introversion Software. *Procedural Content Generation*. 2007. URL: http://www.gamecareerguide.com/features/336/procedural_content_.php (cited on pages 36, 38).

[Joh09] Andy Johnson. *By the Numbers: The Lost Art of Procedural Generation*. 2009. URL: http://www.thegamereviews.com/articlenav-1639-page-1.html (cited on page 40).

[KT98] Marcelo Kallmann and Daniel Thalmann. "Modeling Objects for Interaction Tasks". In: *Proc. Eurographics Workshop on Animation and Simulation*. 1998, pages 73–86 (cited on page 34).

[KMN88] Alex D. Kelley, Michael C. Malin, and Gregory M. Nielson. *Terrain simulation using a model of stream erosion*. Volume 22. 4. ACM, 1988, pages 263–268. ISBN: 0897912756. DOI: 10.1145/378456.378519 (cited on page 44).

[KM06] George Kelly and Hugh McCabe. "A survey of procedural techniques for city generation". In: *ITB Journal* 14 (2006), pages 87–130 (cited on pages 43, 46, 47).

[KM07] George Kelly and Hugh McCabe. "Citygen: An Interactive System for Procedural City Generation". In: GDTW '07 (2007). URL: http://www.citygen.net/files/citygen_gdtw07.pdf (cited on pages 49, 51, 67).

[KW11] Tom Kelly and Peter Wonka. “Interactive architectural modeling with procedural extrusions”. In: *ACM Transactions on Graphics* 30.2 (Apr. 2011), pages 1–15. ISSN: 07300301. DOI: 10.1145/1944846.1944854. URL: http://dl.acm.org/citation.cfm?id=1944846.1944854 (cited on pages 55, 56).

[KTB09] Jassin Kessing, Tim Tutenel, and Rafael Bidarra. “Services in Game Worlds: A Semantic Approach to Improve Object Interaction”. In: *Entertainment Computing – ICEC 2009*. Edited by Stéphane Natkin and Jérôme Dupire. Volume 5709. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009, pages 276–281. ISBN: 978-3-642-04051-1. URL: http://dx.doi.org/10.1007/978-3-642-04052-8_33 (cited on page 34).

[KTB12] Jassin Kessing, Tim Tutenel, and Rafael Bidarra. “Designing Semantic Game Worlds”. In: *PCG 2012 - Workshop on Procedural Content Generation for Games*. 2012. URL: http://graphics.tudelft.nl/~rafa/myPapers/bidarra.Tim.PCG12.pdf (cited on page 89).

[KK11] Lars Krecklau and Leif Kobbelt. “Procedural Modeling of Interconnected Structures”. In: *Computer Graphics Forum* 30.2 (Apr. 2011), pages 335–344. ISSN: 01677055. DOI: 10.1111/j.1467-8659.2011.01864.x. URL: http://doi.wiley.com/10.1111/j.1467-8659.2011.01864.x (cited on pages 54, 91, 92, 147, 149).

[KK12] Lars Krecklau and Leif Kobbelt. “Interactive modeling by procedural high-level primitives”. In: *Computers & Graphics* 36.5 (Aug. 2012), pages 376–386. ISSN: 00978493. DOI: 10.1016/j.cag.2012.03.028. URL: http://linkinghub.elsevier.com/retrieve/pii/S0097849312000672 (cited on pages 54, 55, 67, 163).

[KPK10] Lars Krecklau, Darko Pavic, and Leif Kobbelt. “Generalized Use of Non-Terminal Symbols for Procedural Modeling”. In: *Computer Graphics Forum* 29.8 (Dec. 2010), pages 2291–2303. ISSN: 01677055. DOI: 10.1111/j.1467-8659.2010.01714.x. URL: http://doi.wiley.com/10.1111/j.1467-8659.2010.01714.x (cited on pages 53, 54, 84, 91, 92, 147, 149).

[Kro14] Lennart Kroes. *Een procedurele generatiemethode voor het indelen van woonpercelen*. Technical report. TNO, 2014, page 51 (cited on page 137).

[LG06] Mathieu Larive and Veronique Gaildrat. “Wall Grammar for Building Generation”. In: *Proceedings of the 4th International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia*. GRAPHITE ’06. New York, NY, USA: ACM, 2006, pages 429–437. ISBN: 1-59593-564-9. DOI: 10.1145/

1174429.1174501. URL: http://doi.acm.org/10.1145/1174429.1174501 (cited on page 147).

[LD99] Bernd Lintermann and Oliver Deussen. "Modeling of Plants". In: February (1999), pages 2–11 (cited on page 47).

[LWW08] Markus Lipp, Peter Wonka, and Michael Wimmer. "Interactive visual editing of grammars for procedural architecture". In: *ACM Transactions on Graphics* 27.3 (Aug. 2008), page 1. ISSN: 07300301. DOI: 10.1145/1360612.1360701. URL: http://dl.acm.org/citation.cfm?id=1360612.1360701 (cited on pages 54, 163).

[Lop+10] Ricardo Lopes, Tim Tutenel, Ruben M Smelik, Klaas Jan De Kraker, Rafael Bidarra, and The Hague. "A constrained growth method for procedural floor plan generation". In: *Proc 11th Int Conf* 1996 (2010), pages 13–23. URL: http://graphics.tudelft.nl/~rval/papers/lopes.GAMEON10.pdf (cited on page 102).

[Loy02] Jim Loy. *A Tour of the Mandelbrot Set*. 2002. URL: http://www.jimloy.com/fractals/mandel0.htm (cited on page 35).

[MP06] Dean Macri and Kim Pallister. *Procedural 3D Content Generation*. 2006. URL: http://web.archive.org/web/20060719005301/www.intel.com/cd/ids/developer/asmo-na/eng/20247.htm (cited on page 38).

[Man83] Benoit B. Mandelbrot. *The Fractal Geometry of Nature*. 1983. DOI: 10.1119/1.13295 (cited on pages 35, 43).

[MCS11] Tiago Martins, António Coelho, and Pedro Brandão Silva. *Ontologia Urbana para Ambiente Virtual Colaborativo no contexto do Planeamento e Gestão Municipais*. Portuguese. Porto, 2011 (cited on page 42).

[Mar+12] Tiago Martins, Pedro Brandão Silva, António Coelho, and A Augusto Sousa. "An Urban Ontology To Generate Collaborative Virtual Environments For Municipal Planning And Management". In: *GRAPP 2012*. 2012, pages 1–4 (cited on pages 26, 42, 102, 134).

[Mar96] Paul Martz. *Generating Random Fractal Terrain*. 1996. URL: http://www.gameprogrammer.com/fractal.html (cited on page 43).

[Mel+11] Miguel Melo, Pedro Brandão Silva, Luís Magalhães, Maximino Bessa, J.P. Moura, Artur Rocha, Fernando Nunes Ferreira, J.B. Cruz, and A.Augusto Sousa. "3DWikiU-3D Wiki For Urban Environments". In: *SIACG 2011*. 2011, pages 1–4. URL: http://www.fe.up.pt/si/publs_pesquisa.FormView?P_ID=29789 (cited on pages 26, 42).

[Mer07] Paul Merrell. *Example-based model synthesis*. Seattle, Washington, 2007. DOI: http://doi.acm.org/10.1145/1230100.1230119 (cited on pages 56, 57).

[MM08] Paul Merrell and Dinesh Manocha. "Continuous model synthesis". In: *ACM Transactions on Graphics*. Volume 27. 5. ACM, 2008, page 1. ISBN: 0730-0301. DOI: 10.1145/1409060.1409111 (cited on page 56).

[MM11] Paul Merrell and Dinesh Manocha. "Model synthesis: A general procedural modeling algorithm". In: *IEEE Transactions on Visualization and Computer Graphics* 17.6 (2011), pages 715–728. ISSN: 10772626. DOI: 10.1109/TVCG.2010.112 (cited on page 56).

[MG06] Alexandre Meyer and Christophe Godin. "A Survey of Computer Representations of Trees for Realistic and Efficient Rendering". In: *Complexity* (2006). URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.104.5189 (cited on page 48).

[Mil86] Gavin S P Miller. "The definition and rendering of terrain maps". In: *ACM SIGGRAPH Computer Graphics* 20.4 (1986), pages 39–48. ISSN: 00978930. DOI: 10.1145/15886.15890 (cited on page 43).

[M+06] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, L. Van Gool, and Luc Van Gool. *Procedural Modeling of Buildings*. Volume 25. 3. Boston, Massachusetts: ACM, 2006, pages 614–623. DOI: http://doi.acm.org/10.1145/1179352.1141931. URL: http://dl.acm.org/citation.cfm?id=1141931 (cited on pages 41, 51, 53, 60, 62, 67, 87, 91, 92, 117, 147, 148).

[M+07] Pascal Müller, Gang Zeng, Peter Wonka, Luc Van Gool, and Luc Van Gool. "Image-based Procedural Modeling of Facades". In: *ACM SIGGRAPH 2007 papers on - SIGGRAPH '07* 1 (2007), page 85. DOI: 10.1145/1275808.1276484. URL: http://portal.acm.org/citation.cfm?doid=1275808.1276484 (cited on pages 41, 61).

[MKM89] F. Kenton Musgrave, Craig E. Kolb, and Robert S. Mace. "The synthesis and rendering of eroded fractal terrains". In: *ACM SIGGRAPH Computer Graphics* 23.3 (1989), pages 41–50. ISSN: 00978930. DOI: 10.1145/74334.74337 (cited on page 44).

[NWD05] Benjamin Neidhold, Markus Wacker, and Oliver Deussen. "Interactive physically based fluid and erosion simulation". In: *Natural Phenomena* (2005), pages 25–32. ISSN: 18160867 (cited on page 44).

[Ols04] Jacob Olsen. “Realtime procedural terrain generation”. In: *Department of Mathematics And Computer Science (. . .* (2004), page 20. DOI: 10.1.1.366.6507 (cited on pages 44, 45).

[Opp86] Peter E. Oppenheimer. “Real time design and animation of fractal plants and trees”. In: *ACM SIGGRAPH Computer Graphics*. Volume 20. 4. ACM, 1986, pages 55–64. ISBN: 0897911962. DOI: 10.1145/15886.15892 (cited on page 46).

[Pan04] Pandromeda Inc. *Mojoworld*. 2004. URL: http://www.pandromeda.com/products/ (cited on page 44).

[PM01] Yoav I. H. Parish and Pascal Müller. “Procedural Modeling of Cities”. In: *28th annual conference on Computer graphics and interactive techniques* August (2001), pages 301–308. DOI: 10.1145/383259.383292. URL: http://portal.acm.org/citation.cfm?doid=383259.383292 (cited on pages 42, 49–51).

[Pat12] Gustavo Patow. “User-Friendly Graph Editing for Procedural Modeling of Buildings”. In: *IEEE Computer Graphics and Applications* April (2012). URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5590230 (cited on pages 55, 67, 148, 149).

[PCG09a] PCG Wiki. *Procedural Generation*. 2009. URL: http://pcg.wikidot.com/pcg-algorithm:procedural-generation (cited on page 36).

[PCG09b] PCG Wiki. *What PCG Is*. 2009. URL: http://pcg.wikidot.com/what-pcg-is (cited on page 36).

[Per85] Ken Perlin. “An image synthesizer”. In: *ACM SIGGRAPH Computer Graphics* 19.3 (1985), pages 287–296. ISSN: 00978930. DOI: 10.1145/325165.325247 (cited on page 43).

[Pey+09] Adrien Peytavie, Eric Galin, Jerome Grosjean, and Stephane Merillou. “Arches: A framework for modeling complex terrains”. In: *Computer Graphics Forum*. Volume 28. 2. Wiley Online Library, 2009, pages 457–467. ISBN: 1467-8659. DOI: 10.1111/j.1467-8659.2009.01385.x (cited on pages 43, 45).

[Pla09] Planetside Software. *Planetside*. 2009. URL: http://www.planetside.co.uk/ (cited on pages 43, 44).

[PL96] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1996 (cited on page 47).

[Rem08] Chris Remo. *MIGS: Far Cry 2’s Guay On The Importance Of Procedural Content*. 2008. URL: http://www.gamasutra.com/php-bin/news_index.php?story=21165 (cited on pages 37, 38).

[RP04] Timothy Roden and Ian Parberry. “From Artistry to Automation: A Structured Methodology for Procedural Content Creation”. In: *Entertainment Computing – ICEC 2004*. Edited by Matthias Rauterberg. Volume 3166. Springer Berlin / Heidelberg, 2004, pages 301–304. DOI: 10.1007/978-3-540-28643-1_19. URL: http://dx.doi.org/10.1007/978-3-540-28643-1_19 (cited on page 36).

[Sch10] Stephen Schmitt. *World Machine*. 2010. URL: http://www.world-machine.com (cited on page 44).

[SBW06] Jens Schneider, T. Boldte, and Rüdiger Westermann. “Real-time editing, synthesis, and rendering of infinite landscapes on GPUs”. In: *Vision, modeling, and visualization 2006: proceedings*. IOS Press, 2006, page 145. ISBN: 3898380815 (cited on page 44).

[SD05] Soner I. Sen and A. M. Day. “Modelling trees and their interaction with the environment: A survey”. In: *Computers and Graphics (Pergamon)* 29.5 (2005), pages 811–823. ISSN: 00978493. DOI: 10.1016/j.cag.2005.08.025 (cited on page 48).

[Sid15] Side Effects Software. *Houdini*. 2015. URL: http://www.sidefx.com/ (visited on 06/01/2015) (cited on pages 55, 96, 148–150).

[SC10] Pedro Brandão Silva and António Coelho. “Procedural Modeling for Realistic Virtual Worlds Development”. In: *SLACTIONS*. Taipei, Taiwan: ACM, 2010. URL: https://journals.tdl.org/jvwr/article/view/2109 (cited on pages 27, 42, 60).

[SC11] Pedro Brandão Silva and António Coelho. “A Procedural Modeling Grammar for Virtual Urban Environment Creation”. In: *paginas.fe.up.pt* (2011). URL: http://paginas.fe.up.pt/~prodei/dsie11/images/pdfs/s4-4.pdf (cited on page 27).

[SCR12] Pedro Brandão Silva, Antonio Coelho, and Rosaldo J. F. Rossetti. “A Collaborative Environment for Urban Landscape Simulation”. In: *CoMetS 2012*. 2012, pages 256–261. DOI: 10.1109/WETICE.2012.13. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6269738 (cited on page 26).

[Sil+12] Pedro Brandão Silva, António Coelho, Rui Rodrigues, and A Augusto Sousa. “A Procedural Gometry Modeling API”. In: *GRAPP 2012*. 2012, page 6 (cited on page 26).

[Sil+13] Pedro Brandão Silva, Pascal Müller, Rafael Bidarra, and António Coelho. “Node-based Shape Grammar Representation and Editing”. In: *Proceedings of the 2013*

Workshop on Procedural Content Generation in Games - PCG '13. 2013 (cited on pages 26, 147).

[Sil+15] Pedro Brandão Silva, Elmar Eisenmann, Rafael Bidarra, and António Coelho. "Procedural Content Graphs for Urban Modeling". In: *International Journal of Computer Games Technology* 15 (2015), page 15. DOI: 10.1155/2015/808904. URL: http://dx.doi.org/10.1155/2015/808904 (cited on page 25).

[SMS06] Luiz Gonzaga da Silveira, Soraia Raupp Musse, and Luiz Gonzaga da Silveira. "Realtime Generation of Populated Virtual Cities". In: *Proceedings of the ACM symposium on Virtual reality software and technology* (2006), pages 155–164. DOI: http://doi.acm.org/10.1145/1180495.1180527. URL: http://portal.acm.org/citation.cfm?doid=1180495.1180527 (cited on pages 42, 49, 58, 62).

[SS00] M. Slater and A. Steed. "A virtual presence counter". In: *Presence: Teleoperators & Virtual Environments* 9.5 (2000), pages 413–434. ISSN: 1054-7460. DOI: 10.1162/105474600566925. URL: http://discovery.ucl.ac.uk/92764/ (cited on page 30).

[Sme+10] Ruben Smelik, Tim Tutenel, Klaas Jan de Kraker, and Rafael Bidarra. "Integrating procedural generation and manual editing of virtual worlds". In: *Proceedings of the 2010 Workshop on Procedural Content Generation in Games - PCG '10*. New York, New York, USA: ACM Press, June 2010, pages 1–8. ISBN: 9781450300230. DOI: 10.1145/1814256.1814258. URL: http://dl.acm.org/citation.cfm?id=1814256.1814258 (cited on page 57).

[Sme+09] Ruben M. Smelik, Klaas Jan De Kraker, Saskia a. Groenewegen, Tim Tutenel, and Rafael Bidarra. "A survey of procedural methods for terrain modelling". In: *3AMIGAS - 3D Advanced Media In Gaming And Simulation*. 2009, pages 25–34. ISBN: 9781450300230. DOI: 10.1145/1814256.1814258. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.186.420\&rep=rep1\&type=pdf (cited on pages 43, 44).

[Sme+11] Ruben M. Smelik, Tim Tutenel, Klaas Jan De Kraker, and Rafael Bidarra. "A declarative approach to procedural modeling of virtual worlds". In: *Computers and Graphics (Pergamon)* 35.2 (Apr. 2011), pages 352–363. ISSN: 00978493. DOI: 10.1016/j.cag.2010.11.011. URL: http://linkinghub.elsevier.com/retrieve/pii/S0097849310001809 (cited on pages 40, 57, 163).

[sof11] E on software. *Vue 9.5 Infinite*. 2011. URL: http://www.e-onsoftware.com/products/vue/vue_9.5_infinite/ (cited on page 44).

[Sti80] George Stiny. “Introduction to shape and shape grammars”. In: *Environment and Planning B: Planning and Design* 7.3 (1980), pages 343–351. ISSN: 0265-8135. DOI: 10.1068/b070343. URL: http://www.envplan.com/abstract.cgi?id=b070343 (cited on pages 52, 60).

[SG72] George Stiny and James Gips. “Shape grammars and the generative specification of painting and sculpture”. In: *Information Processing 71 Proceedings of the IFIP Congress 1971. Volume 2*. Edited by C V Friedman. Volume 71. 1972, pages 1460–1465. ISBN: 0 7204 2063 6. DOI: citeulike-article-id:1526281. URL: <GotoISI>://INSPEC:466862 (cited on pages 52, 60).

[SJ04] Penelope Sweetser and Daniel M Johnson. “Player-Centred Game Environments: Assessing Player Opinions, Experiences and Issues”. In: Proceedings Entertainment Computing - ICEC 2005, Lecture Notes in Computer Science 3166 (2004), pages 321–332 (cited on page 30).

[Tak09] Take-Two Interactive Software. *Borderlands*. 2009. URL: http://www.borderlandsthegame.com (cited on page 40).

[.th04] .theprodukkt. *Kkrieger*. 2004. URL: http://www.theprodukkt.com/kkrieger (cited on pages 40, 41).

[Tog+10] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. “Search-based procedural content generation”. In: *Applications of Evolutionary Computation* (2010), pages 141–150 (cited on pages 36, 37).

[Tot09] Total Spore. *Spore: Procedural Generation*. 2009. URL: http://totalspore.com/game-information/procedural-generation/ (cited on page 40).

[Tut+08] Tim Tutenel, Rafael Bidarra, Ruben M. Smelik, and Klaas Jan De Kraker. “The role of semantics in games and simulations”. In: *Computers in Entertainment* 6.4 (2008), page 1. ISSN: 15443574. DOI: 10.1145/1461999.1462009 (cited on pages 34, 39).

[Tut+09] Tim Tutenel, Ruben M Smelik, Rafael Bidarra, and Klaas Jan de Kraker. “Using Semantics to Improve the Design of Game Worlds”. In: *Fifth Artificial Intelligence for Interactive Digital Entertainment Conference*. 2009, pages 100–105. ISBN: 9781577354314. URL: http://www.aaai.org/ocs/index.php/AIIDE/AIIDE09/paper/viewFile/805/1083 (cited on pages 39, 40).

[Tut+11] Tim Tutenel, Ruben M. Smelik, Ricardo Lopes, Klaas Jan de Kraker, and Rafael Bidarra. “Generating Consistent Buildings: A Semantic Approach for Integrating Procedural Techniques”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 3.3 (Sept. 2011), pages 274–288. ISSN: 1943-068X. DOI: 10.

1109/TCIAIG.2011.2162842. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5960781 (cited on page 102).

[VAB12] Carlos A. Vanegas, Daniel G. Aliaga, and Bedrich Benes. "Automatic extraction of Manhattan-World building masses from 3D laser range scans". In: *IEEE Transactions on Visualization and Computer Graphics* 18.10 (Jan. 2012), pages 1627–1637. ISSN: 10772626. DOI: 10.1109/TVCG.2012.30. URL: http://www.ncbi.nlm.nih.gov/pubmed/22291152 (cited on pages 57, 58).

[Vos85] Richard F. Voss. "Random Fractal Forgeries". In: *Fundamental Algorithms for Computer Graphics SE - 34* 17 (1985), pages 805–835. ISSN: 02581248. DOI: 10.1007/978-3-642-84574-1_34 (cited on page 43).

[Wat+08] Benjamin Watson, Pascal Müller, Oleg Veryovka, Andy Fuller, Peter Wonka, and Chris Sexton. "Procedural Urban Modeling in Practice". In: *IEEE Computer Graphics and Applications* 28 (2008), pages 18–26. URL: http://doi.ieeecomputersociety.org/10.1109/MCG.2008.58 (cited on pages 36, 41).

[WP95] Jason Weber and Joseph Penn. "Creation and rendering of realistic trees". In: *SIGGRAPH '95 - Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*. ACM, 1995, pages 119–128. ISBN: 0897917014. DOI: 10.1145/218380.218427 (cited on page 46).

[Wes08] Mick West. *Random Scattering: Creating Realistic Landscapes*. 2008. URL: http://www.gamasutra.com/view/feature/1648/random_scattering_creating_.php (cited on page 38).

[Wil12] Jonathan Williamson. *Tweaking Blender's Interface with Scripting*. 2012. URL: http://cgcookie.com/blender/2012/09/30/blender-tweak-interface-scripting/ (visited on 01/25/2015) (cited on page 144).

[WS94] Bob Witmer and Michael Singer. "Measuring immersion in virtual environments". In: *US Army Res. Inst., Alexandria, VA, Tech. Rep* 1 (1994) (cited on page 30).

[Won+03] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. "Instant architecture". In: *ACM SIGGRAPH 2003 Papers*. Volume 22. 3. San Diego, California: ACM, 2003, pages 669–677. ISBN: 1-58113-709-5. DOI: http://doi.acm.org/10.1145/1201775.882324. URL: http://dl.acm.org/citation.cfm?id=882324 (cited on pages 41, 51, 52, 91, 147).

[YK12] DuMim Yoon and Kyung-joong Kim. "3D Game Model and Texture Generation Using Interactive Genetic Algorithm". In: *Proceedings of the Workshop at SIGGRAPH Asia*. WASA '12 Figure 1. New York, NY, USA: ACM, 2012, pages 53–

58. ISBN: 978-1-4503-1835-8. DOI: 10.1145/2425296.2425305. URL: http://doi.acm.org/10.1145/2425296.2425305 (cited on page 147).

[You09a] Shamus Young. *Fuel: Defining Procedural*. 2009. URL: http://www.shamusyoung.com/twentysidedtale/?p=5134 (cited on page 35).

[You09b] Shamus Young. *The Future is Procedural*. 2009. URL: http://www.escapistmagazine.com/articles/view/columns/experienced-points/6418-The-Future-is-Procedural (cited on page 36).

[Zho+07] Howard Zhou, Jie Sun, Greg Turk, and James M Rehg. "Terrain Synthesis from Digital Elevation Models". In: *IEEE Transactions on Visualization and Computer Graphics* 13.4 (2007), pages 834–848. DOI: http://dx.doi.org/10.1109/TVCG.2007.1027 (cited on pages 43, 44).

Annex 1: Construct Tutorial

The Construct, named after the training system used in the Matrix movie, is a procedural generation system, aiming primarily at the automatic production of content for use in games, movies, training, simulations, etc. This content can range from 3D models to textures, sound, music, game objects, etc. It is currently being developed for research in the scope of a Ph.D. project, but could eventually become available for more widespread and professional use. In this sense, you are a pioneer and your feedback would be precious. At the end of this tutorial, we would ask you to answer a quick survey on your experience.

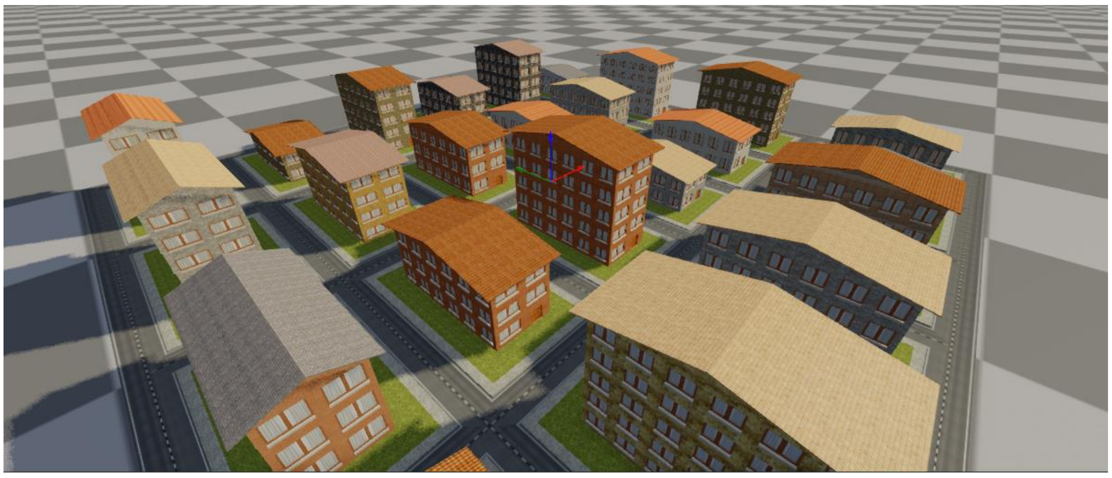


Figure 8.1: Expected final result of this tutorial.

By the end of this document, you should have some basic experience with the Construct and with the concept of procedural generation. You will be introduced to procedural content graphs, a visual approach to define sequences of computer instructions that create content in an algorithmic way. We will focus on the generation of a 3D model of a neighborhood, as seen in the picture above. To achieve this, we will focus on building small components (door, window, house, etc.) first and then proceed join all them all.

We hope you enjoy working with the Construct!

Installation

The Construct is only available for the Windows platform (no versions prior to Windows 7 have been tested). Also, the Construct requires two essential applications to be installed, namely:

- .NET Framework
  `http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=17851`
- Microsoft XNA Framework Redistributable 4.0
  `http://www.microsoft.com/download/en/details.aspx?id=20914`

If you run into trouble when trying to run the editor, you're probably missing one of these installations. The Construct itself does not require any installation. Unzipping the .zip file to a location with writing permissions is enough.

Assets

For the purpose of this tutorial, we have made a set of textures available:

`http://construct.pabsilva.com/files/Assets.zip`

These and further textures can be obtained from `http://www.cgtextures.com/`

Getting started

Start the Construct by double-clicking the **Construct Editor.exe** file. This may take a few moments, since many plugins and libraries have to be loaded. The following window should then appear:

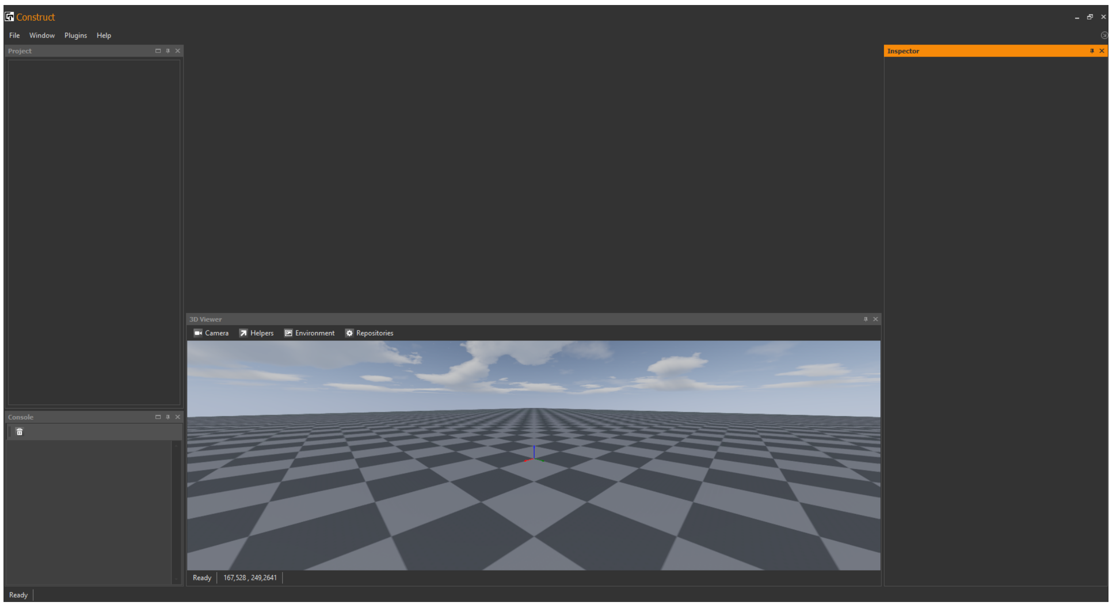


Figure 8.2: Construct Graphics User Interface.

This is the default design layout of the Construct, which is composed by the following tool panels:

- **Project:** Lists the files and folders of the project you're currently working on. In the Construct, you can only work in one project at the time, but you can switch to another project by going to **File->Open Project** or **File->Recent Projects**. Project files are displayed in a tree.
- **Inspector:** Initially empty, but is used to show the properties of items selected in other tool windows.
- **3D Viewer:** Used to display and interact with 2D and 3D content. Camera controls are as follows:
  - Middle mouse button (or left+right mouse button) to pan the camera
  - Mouse wheel to zoom in and out
  - WASD, C and Space keys in your keyboard to control the position of the camera as it would be a first-person shooter. W for front, S for back, A for left, D for right, C for down, Space for up.

  There are a lot of viewing options that can be toggled, so as to give you a better perception of the shown models. Be wary that options such as "Show Scope Information", "Toggle

Edges" and "Show Checkerboard" can have an impact on performance for large amounts of displayed objects. Check the Sky and Ocean Options for some nice effects!

- **Console:** Shows text messages sent by other tool windows.
- **Document Area:** The only empty area in the previous figure. It is reserved for viewing and editing project files. We will manipulate graph files in here.

You can change the overall layout of the tool windows (their location and size) to your liking by dragging them with the left mouse button. They can also be positioned over several monitors, if available. To keep the layout for future Construct executions, go to **Window->Save Layout**. You can overwrite existing layouts by choosing their name from the combo box.

You can also change the overall color set of GUI by going to **Window->Skins** on the top menu and selecting a different skin from the list.

Starting a new Project

Start a new project by going to **File->New Project**. Choose a project name (for instance, "MyProject") and a location. A new folder with that project name will be created within the specified location. By default, the Desktop location is suggested. After you do so, the project tool window will show two automatically generated folders: Graphs and Textures.

Unzip the assets to a location of your liking. Now, at the Construct Project Explorer, right-click the root folder and on the context menu select **Add->Existing Folder** and Contents. Select the extracted asset folder and all of its contents will be added to the project.

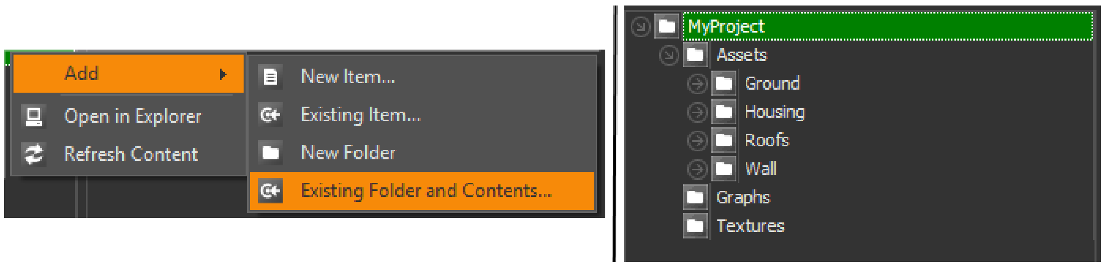


Figure 8.3: Adding assets to the project.

Now, right-click the **Graphs** folder and select **Add->New Item**. On the window that appears, select Graph and name it "House". A light blue window will appear on the document area. Navigation controls are the same as for the 3D Window: Middle mouse button (or left+right mouse button) to pan the camera and mouse wheel to zoom in and out. Use F to zoom of the selected nodes, double-click anywhere to frame the whole graph.

We're set to start working on graphs!

A note on Procedural Content Graphs and Entities

A procedural content graph is a data flow diagram that is used to define procedural generation processes. In other words, it is a visual programming language that uses nodes and edges to define sequences of operations that create content, such as 3D models, images, sound, etc.

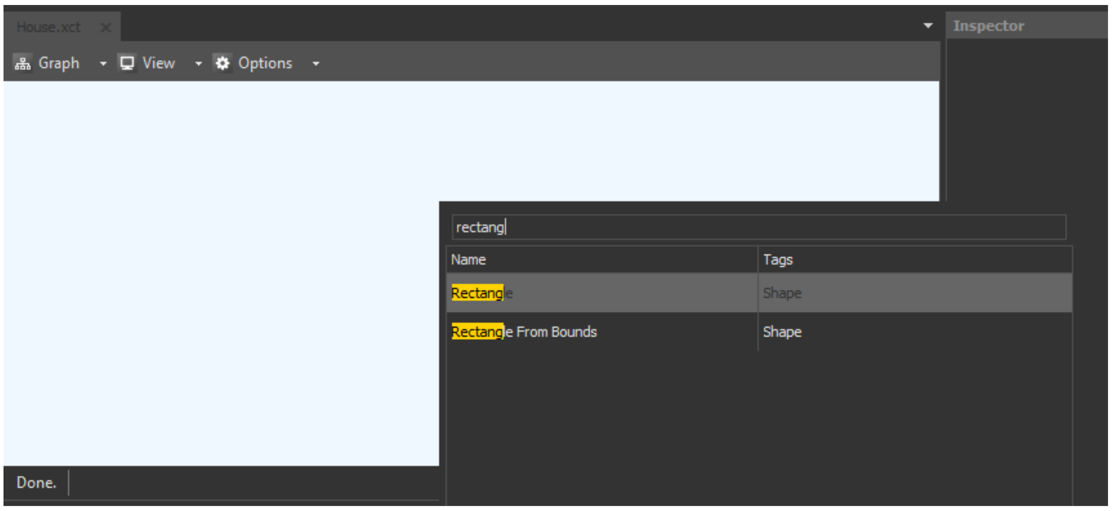
This content is represented in the form of ***Entities***. Entities are sent in and out from nodes through their ***ports***. We're going to deal with two types of entities in this tutorial: Shapes and StreetsNetworks.

- A ***Shape*** is an object that contains a list of geometries. These geometries can be vertices, edges and faces. Each edge connects two vertices. Faces connect several vertices and may even contain holes.
  Each shape contains a ***scope***, which is used to define the size and direction of a shape. It is displayed as a set of 3 arrows, defining the X, Y, and Z direction. Scopes are very important because they allow shapes to have their own reference system.

- A ***StreetNetwork*** is an object that contains a list of street vertices and street edges. They contain information on how streets are connected, similar to how Google Maps presents their maps.

Creating a House: Managing Nodes, Edges and Scopes

In the newly create "House" graph editor area, right-click anywhere in the blue area and, in the context menu that appears, click on **Add Node**. A window with a list of names will appear. This is the list of available nodes, which represent operations. This list can also be shown when the keyboard combination Ctrl + Space is pressed.



rectang

Name	Tags
Rectangle	Shape
Rectangle From Bounds	Shape

Figure 8.4: Searching for an operation in the node list.

Start typing in "Rectangle" in the textbox of the window. As you type, the contents of the list will be filtered to match the name you typed. Once you have the "Rectangle" operation selected, press Enter (or double-click it). The window will close and a new node, called "Rectangle" will appear on the graph.

This node has one ***output port*** (the circle on the right of the node), from which a shape object, containing one rectangular face will come out. Let's view it! Right-click anywhere in the graph area and select the ***Execute*** option. Doing this will execute the graph, which in turn will produce content that will be shown on the 3D Viewer panel.

Because the rectangle is small, it might be difficult to see. On the 3D Viewer menu, go to ***Camera->Frame All*** to zoom in on the create object.

Now click on the node with the left mouse button and a couple of properties will appear on the inspector panel. This allows us to configure the node label and how it node works. In the parameter section, you can change values for "Width" and "Height". These ***parameters*** control the size of the rectangle that is being produced. Change the values to 10 and 15, respectively, and execute again. Every time you execute, the previous content will be thrown away and a new version will be created. You'll see that, in this case, a larger rectangle was created.

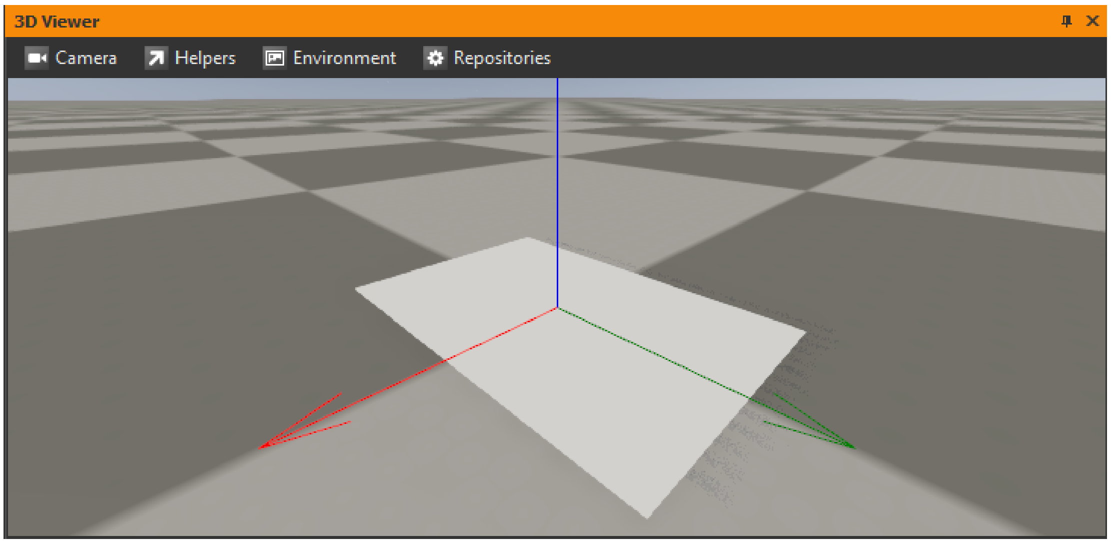


Figure 8.5: Generation of a larger rectangle.

Now add a second node of type "Extrude" using the same approach as before. This new node has one input port (the circle on the left of the node) and one output port. This operation will take all shapes that arrive at its input port, extrude the containing faces (i.e. create a block out of them) and return the new shape.

Let's now connect the output port of the rectangle node to the input port of the extrude node. To do so, move the mouse cursor to the output port of the first node and a small box will tell the name and type of the output. Also, the input port of the second node will be shown in green, meaning that it is compatible (both are of type ***Shape***).

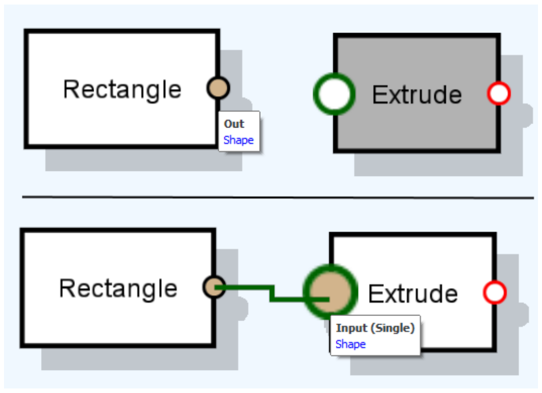
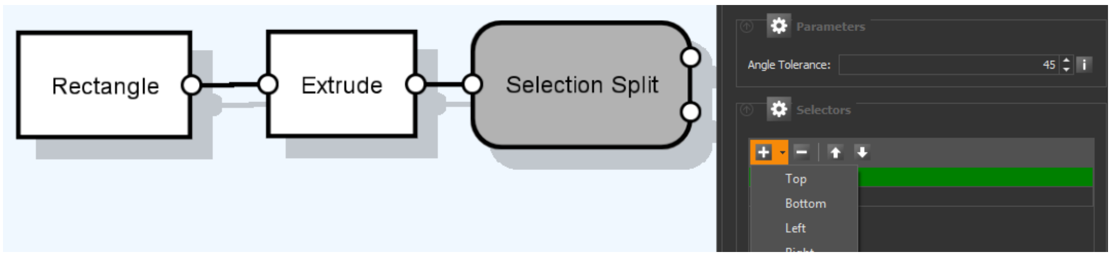


Figure 8.6: Connecting two nodes.

Notice that the output ports of the extrude node are red, because you can't connect an output port to another output port. Only connections of between input and output ports are allowed and they have to be of the same type. Ports of type ***Entity*** accept any kind of data. To create an edge, press and hold the left mouse button down on the starting port and move to the target port. An edge will be drawn as you move the mouse. Release the mouse button on the target port to establish a connection.

Now that this connection exists, the rectangle that is being produced by the "Rectangle" node is passed to the extrude node, which in turn will operate on it and return the modified shaped. Click on the "Extrude" node and set its parameter "Amount" to 10. Execute the graph and see the result - a box.

Let's add a new kind of node - the "Selection Split" - and connect the output port of the "Extrude" to the input of this one. Now click on it. You can see that it has a section called "Selectors". This node allows us to divide our shape block into multiple parts. Using the "+" sign, let's add the "Top" and "All" items to the list. These items are called ***Augmentations***, because they can add more ports, parameters and attributes (more on this later) to the node.



Angle Tolerance: 45

Figure 8.7: Selecting items from the list.

By doing so, we are dividing the top face from the rest (the "all" refers to all that is remaining). Notice that two output ports are added to the node. Each part is stored inside a new shape object and each has now its own scope. This gives the possibility to do different things with each of the parts.

Execute the graph, frame the result and make sure that the scopes are visible (in the 3D

Viewer menu, go to ***Helpers->Show Scope Information***) and the edges as well (go to ***Helpers -> Toggle Edges***). As you can see in Figure 8, we have now two shape objects, each with a different scope.

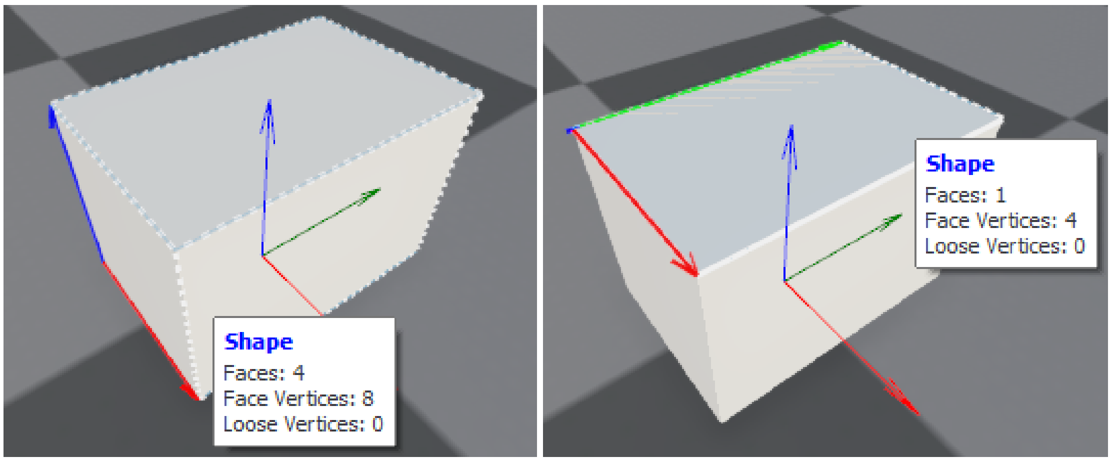


Figure 8.8: Analyzing Scopes of two shapes.

The scopes are displayed by the sets of 3 arrows (red is X, green is Y, blue is Z) and the dotted edges. The default XYZ arrows in the middle show the normal coordinate system axes. On the left, we have our shape that contains the 4 side faces. Its scope has the same direction as the normal coordinate system axes.

Look for a node called "Shed Roof", add it to the graph, connect its input port to the first output port of the "Selection Split" and set the parameters "Height" to 2 and both "Overhang Sizes" to 1. This will take out top face and create a roof out of it. Feel free to try out other values for these parameters to see the effect.

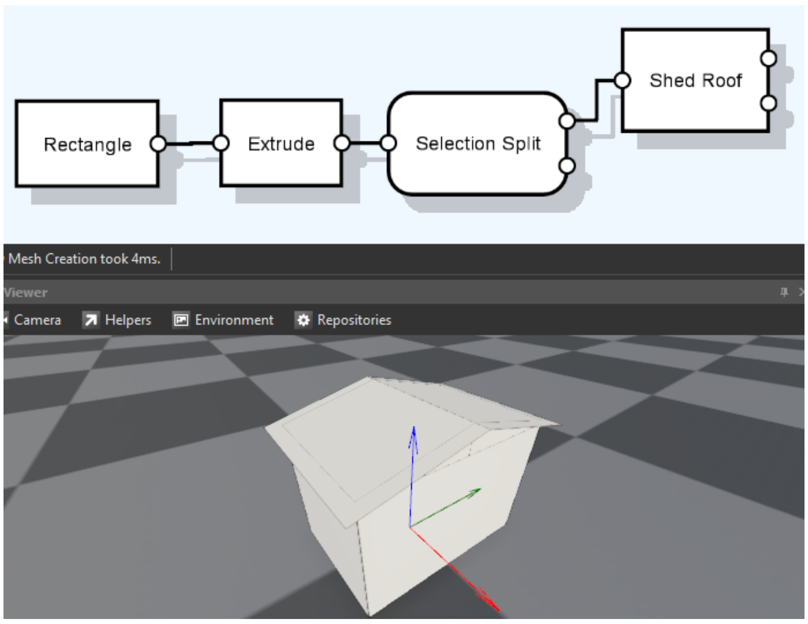


Figure 8.9: Adding a roof to the block.

The roof looks like a piece of paper though. Add a "Special Extrude" node, connect to the first port of the "Shed Roof" and set its parameter "Amount" to 0.1f and check the "Cap". What is the difference between the "Special Extrude" and the "Extrude" nodes? Try the "Extrude" and try to see the difference. What does the "Cap" parameter do? Try deactivating it and try to see the difference!

Let's focus on the facades now. We need to divide our 4-face block shape into individual shapes, each containing independent scopes (and facing different directions). Add the "Decompose" node and connect it to the second output port of the "Selection Split".

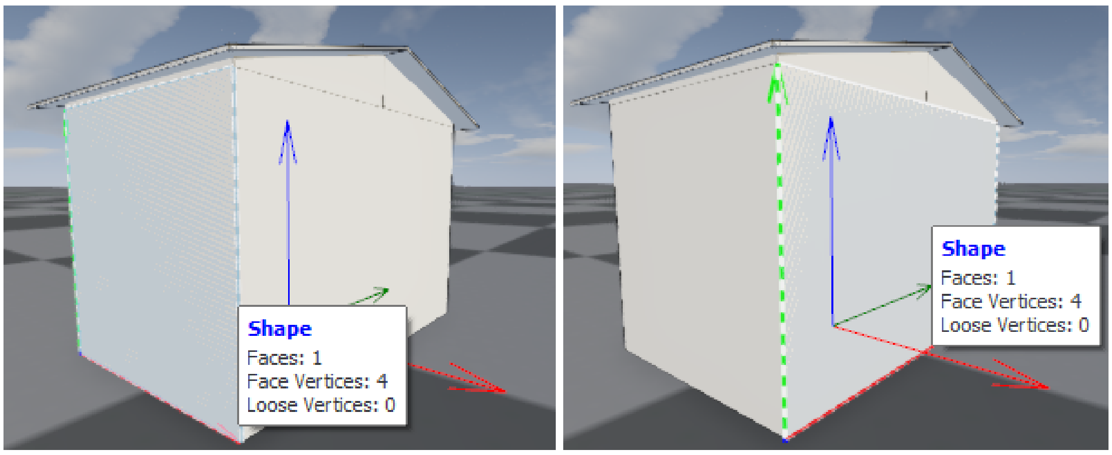
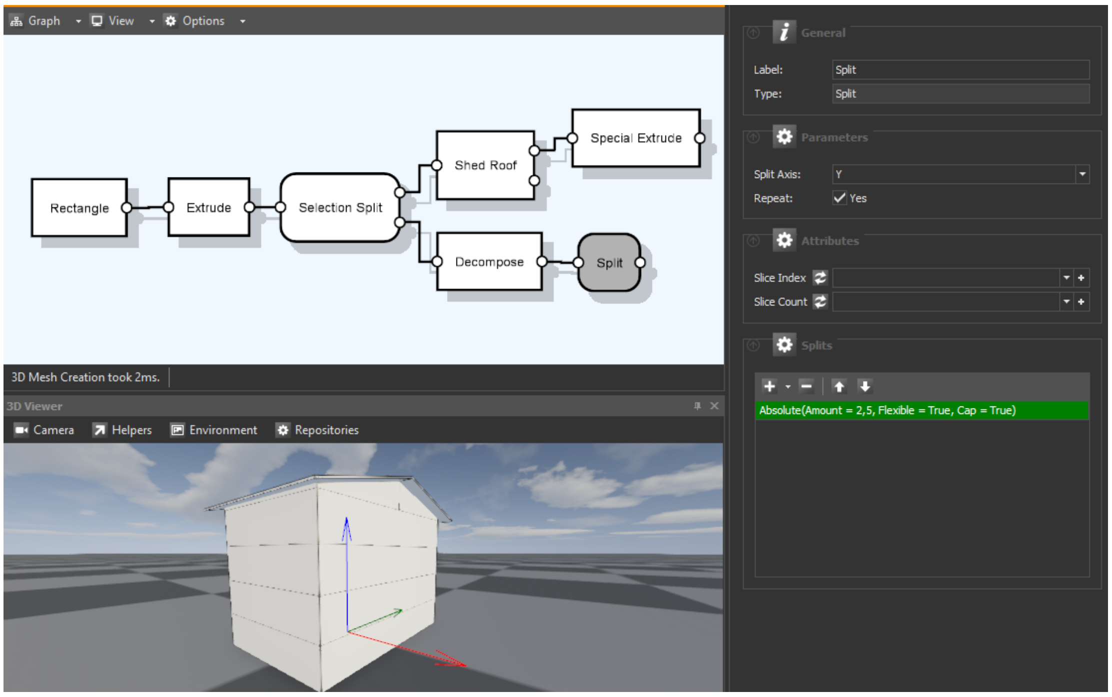


Figure 8.10: Shape decomposition.

Let's add a new node, called "Split" and connect it to the "Decompose" output port. We want to cut each one of our facades into several floors. The "Split" will do just that: for each shape that comes in (remember that the decompose node outputs 4 shapes), the "Split" node will cut it according to a direction in the scope. We will start by cutting in the Y direction (green arrow in Figure 10), so set the parameter "Split Axis" to "Y". Also, we want to cut as many times as possible, so check the "Repeat" option. To define the size of the slices, we can add items to the list in the "Splits" section (again, these are augmentations). Add an item of type "Absolute" and double-click on the item. A popup menu will appear, featuring more parameters. Set 2.5 for the "Amount" parameter. If you execute the graph, you'll see that the faces have all been cut.

Let's now split each floor horizontally in the same way. Right-click the "Split" node, select "Copy" and then "Paste" it just next to it. Connect the output of the existing split to the input of the new split. Now change the parameter "Split Axis" of the new split node to "X". Execute the graph and you'll see that a grid pattern has been created on the facades. We will use this grid to organize the location of the windows and the door.



3D Mesh Creation took 2ms.

Label: Split
Type: Split
Split Axis: Y
Repeat: ✓ Yes
Slice Index
Slice Count
Absolute(Amount = 2,5, Flexible = True, Cap = True)

Figure 8.11: Configuration and result of the split node.

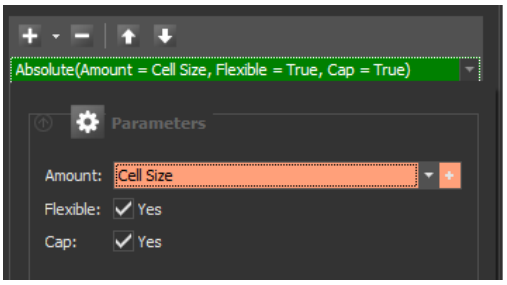
Managing Graph Parameters and Attributes

Before we proceed, let's learn something about global ***parameters*** and ***attributes***. Right-click anywhere in the graph and select "Graph Properties". A new configuration controller appears on the inspector panel. Here we can change the name of the author of this graph and its description. We can also create global parameters: they work as constant values that can be referred to several times across the graph. Click the "+" button to add a new parameter. Double click it and change its name to "Height", Type to "System.Single" and value to 8.

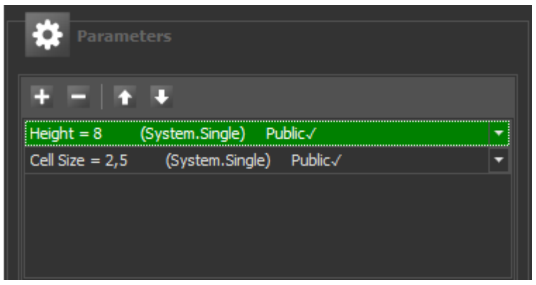
Click back on the "Extrude" and right-click the label of the "Amount" parameter. A context menu will appear, giving us several ***mode*** options. Choose the mode "Global Parameter". The old control will change to a red combo box. Choose the "Height" item from that combo box and execute. The height of the building will change, because it is using the global parameter value (8) instead of the old, fixed value that we had defined before (10).

There are other ways to create and assign global parameters. Go to the first split note and double-click the only item on the list, so as to show the popup. Change the mode of the "Amount" to "Global Parameter" as well. The control will turn red as before. This time, instead of choosing an item from the combo box, click the "+" button next to it. Enter "Cell Size" as a name. This will create a new global parameter that has the original "Amount" value as default (2.5).

Now click at the second split node, double-click the item on the list, change its mode to "Global Parameter" and choose "Cell Size" from the combo box. Now both nodes refer to the same global parameter. This means that, if you change the value at the graph properties, both



```
Absolute(Amount = Cell Size, Flexible = True, Cap = True)
Parameters
Amount: Cell Size
Flexible: ✓ Yes
Cap: ✓ Yes
```

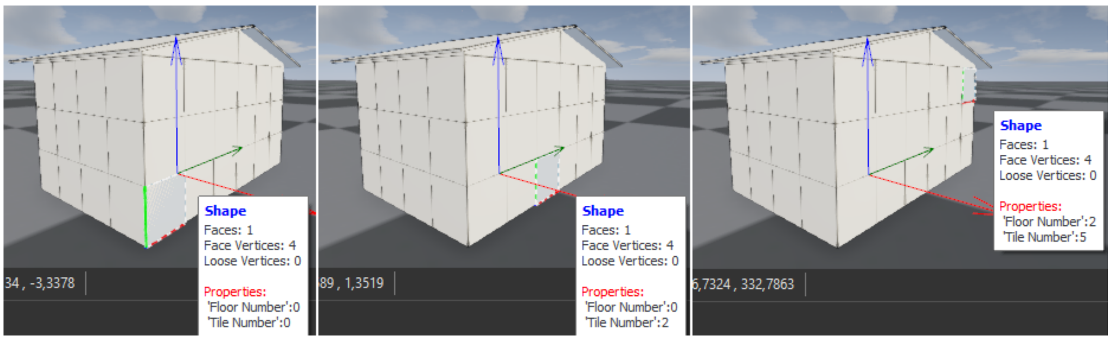


```
Parameters
Height = 8      (System.Single)   Public✓
Cell Size = 2,5   (System.Single)   Public✓
```

Figure 8.12: Creating a global parameter at the split node (Left). List of global parameters at the graph properties (Right).

nodes will be affected. Try changing the value of Cell Size to 2 or 3 and see how it affects the size of the grid cells on the building facades.

Now let’s look into the concept of ***attributes***. Go to the first split node, and on the "Attributes" section, click at the "+" at the far end of the "Slice Index". Enter the name "Floor Number". Now, go to the second split node, and on the "Attributes" section, click at the "+" at the far end of the "Slice Index". Enter the name "Tile Number". Execute the graph.



```
Shape
Faces: 1
Face Vertices: 4
Loose Vertices: 0
Properties:
'Floor Number':0
'Tile Number':0
```

```
Shape
Faces: 1
Face Vertices: 4
Loose Vertices: 0
Properties:
'Floor Number':0
'Tile Number':2
```

```
Shape
Faces: 1
Face Vertices: 4
Loose Vertices: 0
Properties:
'Floor Number':2
'Tile Number':5
```

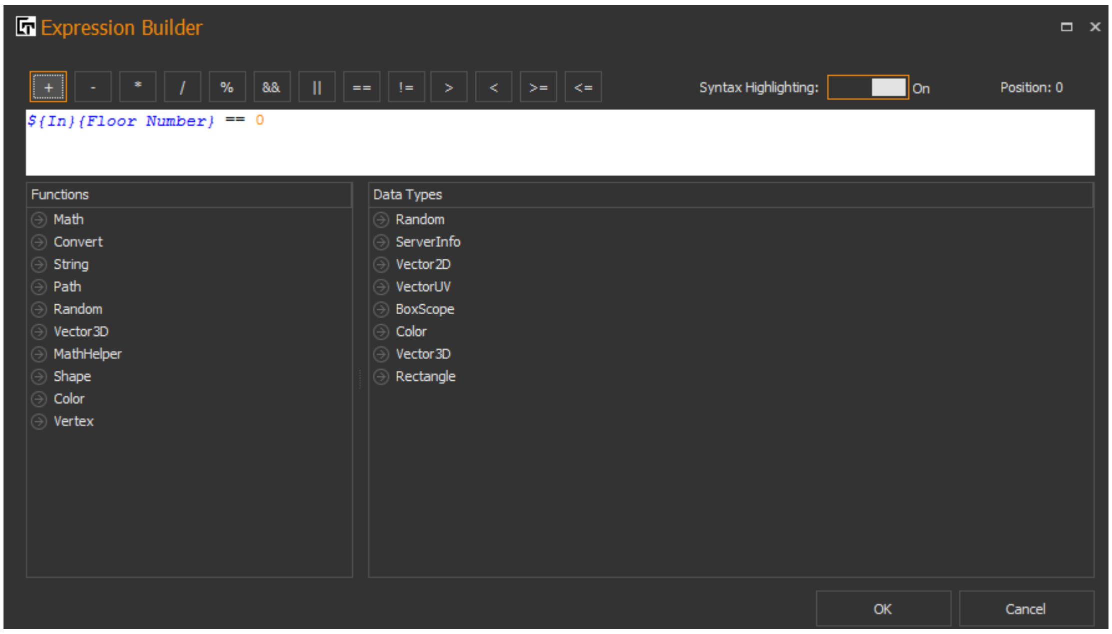
Figure 8.13: Attribute values (section "Properties") for each of the grid tiles.

By hovering each of the grid tiles, you can see that each has a different set of numbers for the "Floor Number" and "Tile Number". And these numbers actually correspond to the floor number (counting from bottom to top, starting at 0) and tile number (counting from left to right, starting at 0). How was this possible?

Well, it so happens that the split node, when cutting a shape, has the ability to give an index/ number to each part. Of course, for us, the first split was to divide the facades into floors, so this number was actually equivalent to the floor number. The second split also had its specific meaning. What we did before in these "Attributes" section was creating two placeholders where to store this numeric information. If you go to the graph properties as before, you’ll see 2 entries on the "Attributes" section.

So, basically, we created 2 attribute definitions, but unlike global parameters, these attributes mean that every entity (here, specifically, shapes) inside this graph have different values associated to this "Floor Number" and "Tile Number". Maybe a bit confusing, but you saw how it was defined and visualized. Let’s learn how to use them.

Add a "Condition" node and connect it to the output port of the second split. For the "Conditional Value" parameter of the node, right-click the label and select the "Expression" mode. It will change to a blue box. Click on the button on the right and an expression editor will appear. Type in the text, as in the image:



```
${In}{Floor Number} == 0
```

Figure 8.14: Expression editor.

What this will do is the following: the shapes that have their attribute "Floor Number" equal zero will be sent out through the first output port of the condition node. The others that do not satisfy this condition will come out of the second output port. To understand this better, add a "Color" node, connect the first output of the "Condition" node to it and execute the graph. You'll see that the bottom floor is all red.

Now, what we want is to select only 1 of these tiles, because we just want to have one door. Add an "Amount Filter" node between the "Condition" and the "Color" so that the first output connects to the "Color" (remove the previous connection). Set the "Amount" parameter to 1 and "Starting Index" to 1. This node filters the indicated amount of objects (starting at the indicated index) that come in and sends them to the first output. The rest is sent to the second output. It is important to notice that this node looks different from previous nodes - the input port is ***square***, not circular like the others. It also works differently - it is a ***gather port***. Until now, all nodes could handle several shapes, but would handle one at the time (for instance, the split takes one shape at the time - it just runs several times). On the other hand, this node will take all shapes in one go and then decides what to do when looking at the whole set.

We'll stop at the House for now and see how to make a window.

Creating a Window

Now, right-click the ***Graphs*** folder and select ***Add->New Item***. On the window that appears, select ***Graph*** and name it "Window".

For the creation of the window, most the operations have already been presented. That being said, we'll just leave the starting graph nodes and some instructions on its parameter settings. Try to understand what each operation is doing and feel free to tune the parameters to your liking.

Since we're going to model a window on an example rectangle, you might want to hide the checkboard platform (***Helpers -> Show Checkerboard platform***). Another useful tip lies on edge disabling: right-click an edge and select ***Disable***. A "forbidden" sign will appear on top of it, signaling that the edge is acting as it wouldn't be there. This is useful to test the impact of a connection without having to actually delete it from the graph.

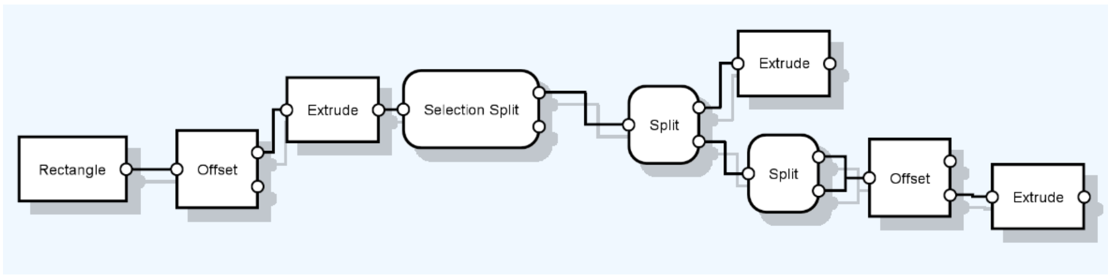


Figure 8.15: Starting Window graph.

- First Rectangle: Width= 3, Height= 3
- First Offset: Amount= -0.7
- First Extrude: Amount= -0.2
- Selection Split:
  - Top
  - All
- First Split: Split Axis=Y, Repeat=false
  - Slice 1 - Absolute: Amount = 0.2, Flexible=false
  - Slice 2 - Absolute: Amount = 0, Flexible=true
- Second Split: Split Axis=X, Repeat=false
  - Slice 1 - **Relative:** Amount = 0.5, Flexible=false
  - Slice 2 - **Relative:** Amount = 0, Flexible=true
- Second Extrude: Amount= 0.3
- Second Offset: Amount= -0.2
- Third Extrude: Amount= 0.1

Once you have reproduced this graph, we can proceed. We'll start by adding some materials to it. We'll need "Texture material" and a "Face UV Mapping". The first node adds a texture to the faces of a shape. The second actually indicates how the texture is applied to the faces (how many times they will repeat). Add one of each and connect them as in the following figure. In the "Texture Material" node, set the "Texture" parameter to "Assets/Housing/Curtains.jpg". In the Face UV mapping, uncheck the "Absolute Sizing".

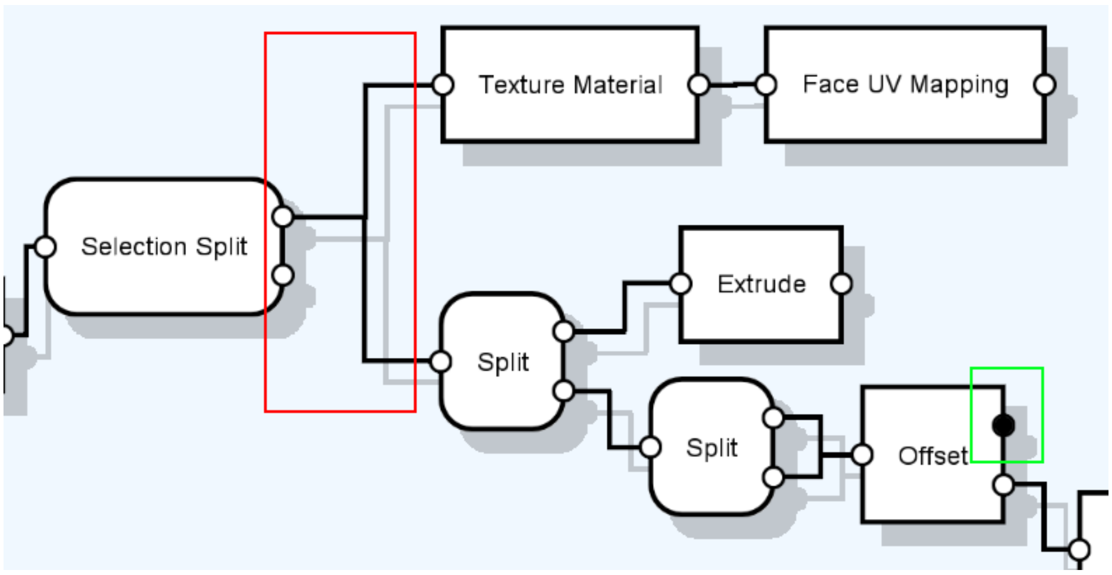


Figure 8.16: Material application.

As highlighted in red in the figure, we have 2 edges coming from the same port. This means that the shapes that come out of that port are being duplicated and each is copy is being handled differently. This allows us to define a background, where we put a curtain texture. On top of it, 2 window doors are created. However, to discard the 3D parts of the generated door interiors, we ***block*** the first output port of the offset (highlighted in green) by left-clicking on the port and selecting the "Blocked" option. Try with and without the blocking to see the difference!

Add 2 new "Texture Material" nodes and connect them to the 2 last extrude node output ports. Set "Assets/Wall/Wall01.jpg" and "Assets/Housing/WoodFine.jpg" as values for the "Texture" field, for texturing the ledge and window frame, respectively. Add a "Face UV Mapping" to the end of each of these 2 "Texture Material" nodes. Now, the created window is composed by many small parts and result in many shape objects. What we can do is to merge them, using the "Merge" node. Add 2 merge nodes and connect them like in the figure below (highlighted). In the end, we'll have only 2 shapes: one for the actual window and another for the surrounding wall.

We now want to use this window in our house. We can transform this big graph into a small node and use it inside our building graph, a process we call ***Encapsulation***. But first, we need to indicate what the inputs and outputs of that node will be. Click the input port of the first "Offset" node and select the option "Gate". In the gate options, write "Tile" for the name. For the first

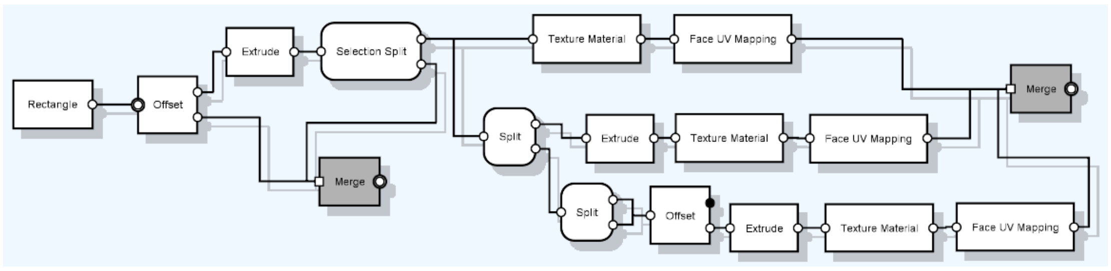


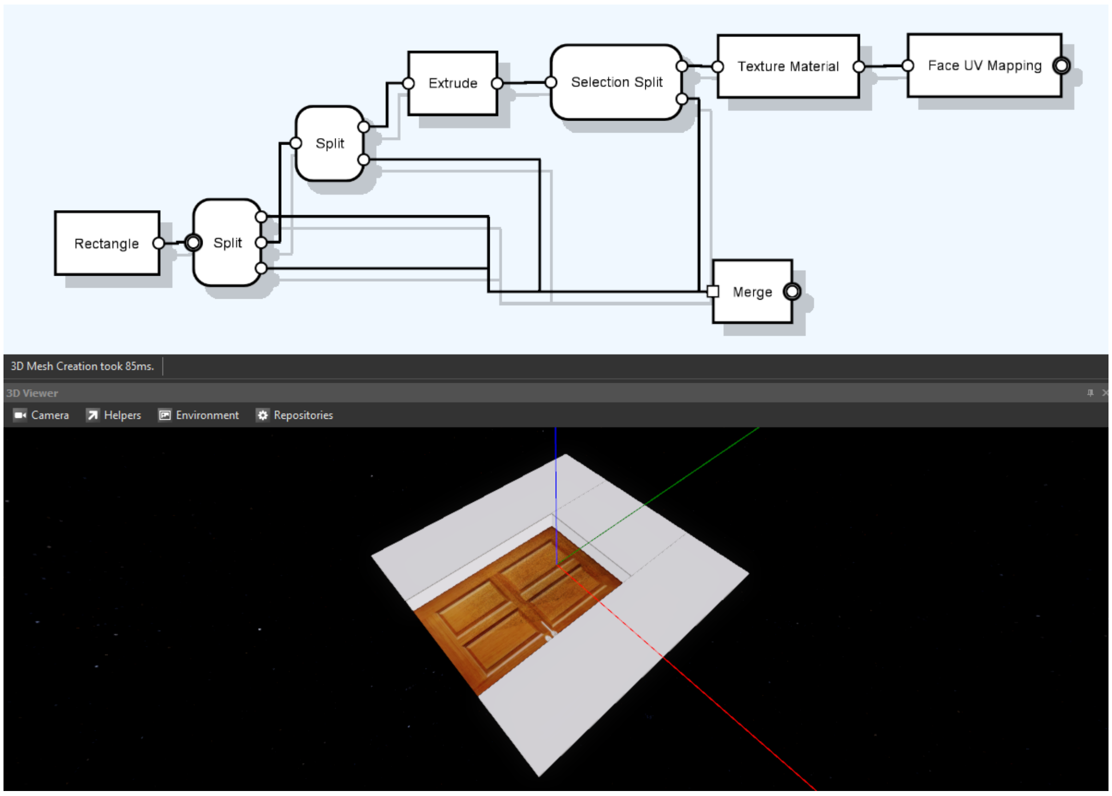
Figure 8.17: Final Window Graph.

merge node from the left, set its output port as a gate as well, but name it "Wall". Finally, for the last merge node, set its output port as a gate, but name it "Window".

Creating a Door

Now, right-click the ***Graphs*** folder and select ***Add->New Item***. On the window that appears, select Graph and name it "Door".

Having explained the basic concepts and show in detail how to make a window, we'll leave just some basic instructions on how to make a door. If a certain parameter value isn't mentioned, it is because the value should remain the default one. Do not forget to mark and name the gates of the graph!



3D Mesh Creation took 85ms.

Figure 8.18: Door Graph.

- Rectangle: Width=3, Height=3
- First Split: Split Axis=X, Repeat=No (input is gate - name it "Tile")
  - Slice 1 - Absolute: Amount = 0, Flexible=true
  - Slice 2 - Absolute: Amount = 1, Flexible=false
  - Slice 3 - Absolute: Amount = 0, Flexible=true
- Second Split: Split Axis=Y, Repeat=No
  - Slice 1 - Absolute: Amount = 2, Flexible=false
  - Slice 2 - Absolute: Amount = 0, Flexible=true
- Extrude: Amount= -0,2
- Selection Split:
  - Top
  - All
- Texture Material: Texture="Assets/Housing/DoorWood.jpg"
- Face UV Mapping (output is gate - name it "Door")
- Merge: (output is gate - name it "Wall")

Assembling the House

Open the house graph again. To use the window and door graphs as nodes in this graph, simply drag and drop the files from the project explorer panel to the graph area. The nodes will appear, with the indicated gates as ports and the global parameters as node parameters (feel free to try adding some!).

Remove the color node and connect the encapsulated nodes as following. Execute the graph and you'll get the following result.

We still have to put some textures on the roof and walls. Do the usual "Texture material" + "Face UV Mapping" combination for the roof, but set the texture name as a global parameter (name it "Roof Texture"). As default, choose one texture from the "Assets/Roofs" folder. As for the walls, connect the ports called "Wall" from the "Door" and "Window" nodes, together with the second output port of the "Shed Roof" to a new "Merge" node. This should be followed by an "Align Scope to Axes", "Box UV Mapping" and "Texture Material", as in the following figure. Again, set the texture name as a global parameter (name it "Wall Texture"). As default, choose one texture from the "Assets/Wall" folder.

Before leaving, mark the input port of the first extrude node as a gate. Also, go to the ***Graph Properties -> Output Style*** and choose "Aggregative". This will be explained further on.

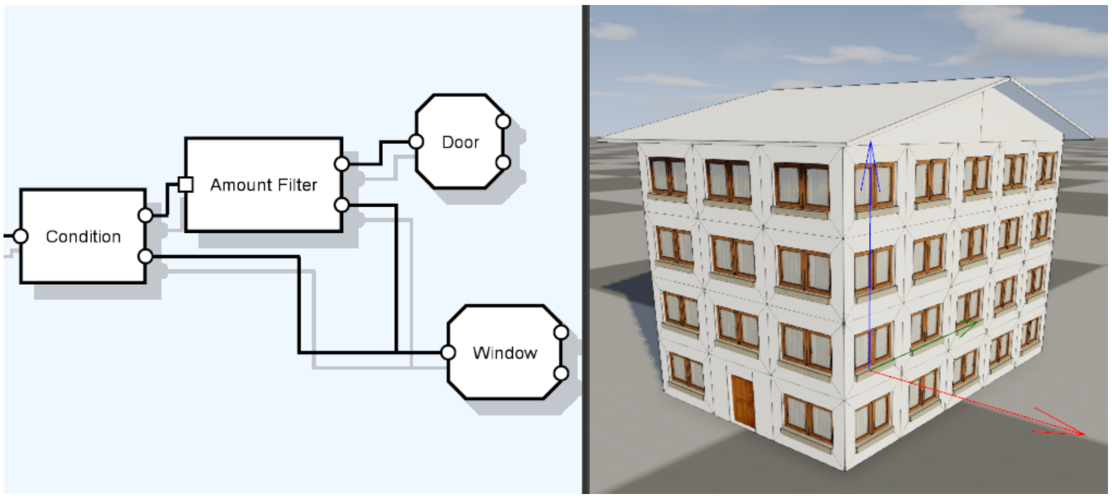


Figure 8.19: Windows and Door added to the building block.

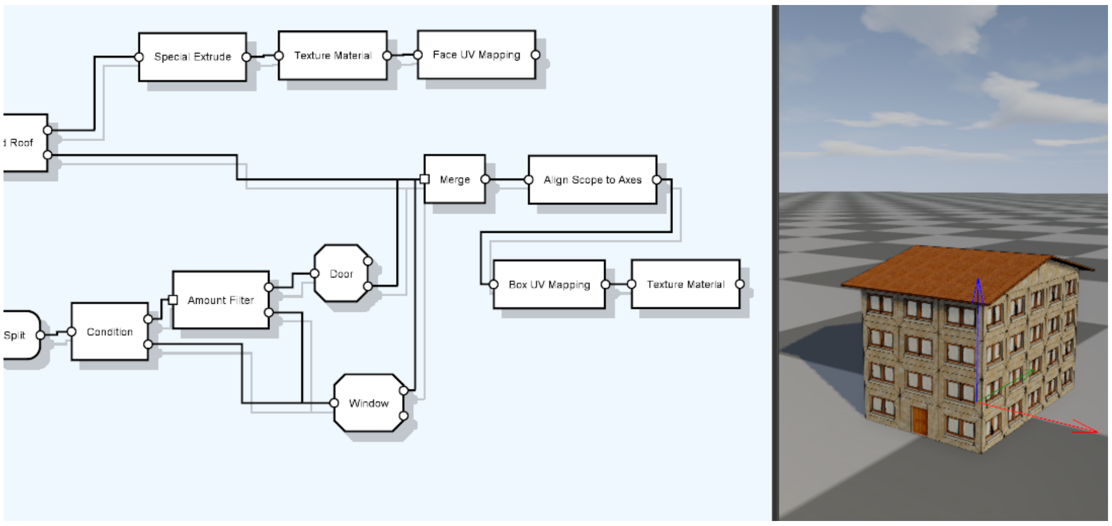


Figure 8.20: Building with materials.

Building the Street Network

Now, right-click the Graphs folder and select ***Add->New Item***. On the window that appears, select ***Graph*** and name it "Streets". As before, try to recreate the graph below, using the given instructions.

- Rectangle: Width=80, Height=80
- First Split: Split Axis=X, Repeat=Yes
  - Slice 1 - Absolute: Amount = 20, Flexible=true
- Second Split: Split Axis=Y, Repeat=Yes
  - Slice 1 - Absolute: Amount = 20, Flexible=true
- Street to Shape: Width=4 (second output is gate - name it "Lots")
- Texture Material: Texture="Assets/Ground/Roads.jpg"

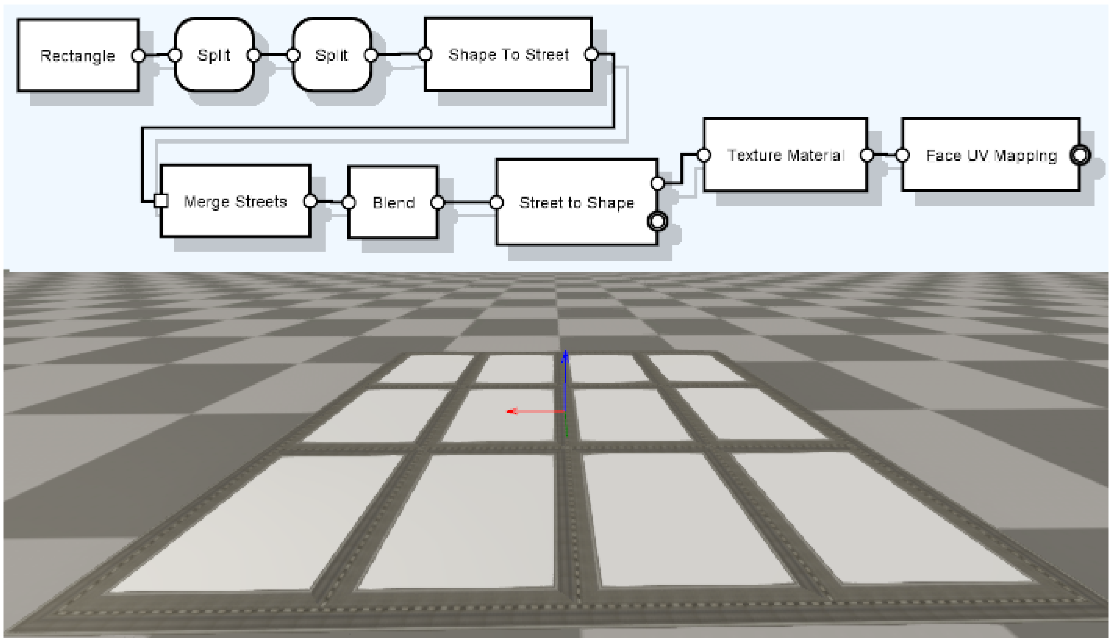


Figure 8.21: Street and Lot generation.

- Face UV Mapping: UV = 4,2 (output is gate - name it "Streets")

So, what are we doing here? Basically we're creating a grid pattern on the rectangle (as before) and we're getting one shape per tile. Afterwards, we transform the boundaries of the faces in these shapes into streets (using the shape to street). But we have to merge the streets and blend them, in order for the crossings to be found. After that, we convert the street back to shapes using the "Street to Shape" operation. This node returns some nice streets (which we are texturing) and some nice lots in between. We can put out houses here now.

Building the Neighborhood

Now, right-click the Graphs folder and select ***Add->New Item***. On the window that appears, select Graph and name it "Streets". As before, try to recreate the graph below, using the given instructions.

- Streets: (just drag and drop, as before)
- Extrude (connected to the "Lots" port): Amount=0.2
- Selection Split:
  - Top
  - All
- First Offset: Amount=-1.5
- Second Offset: Amount=-4

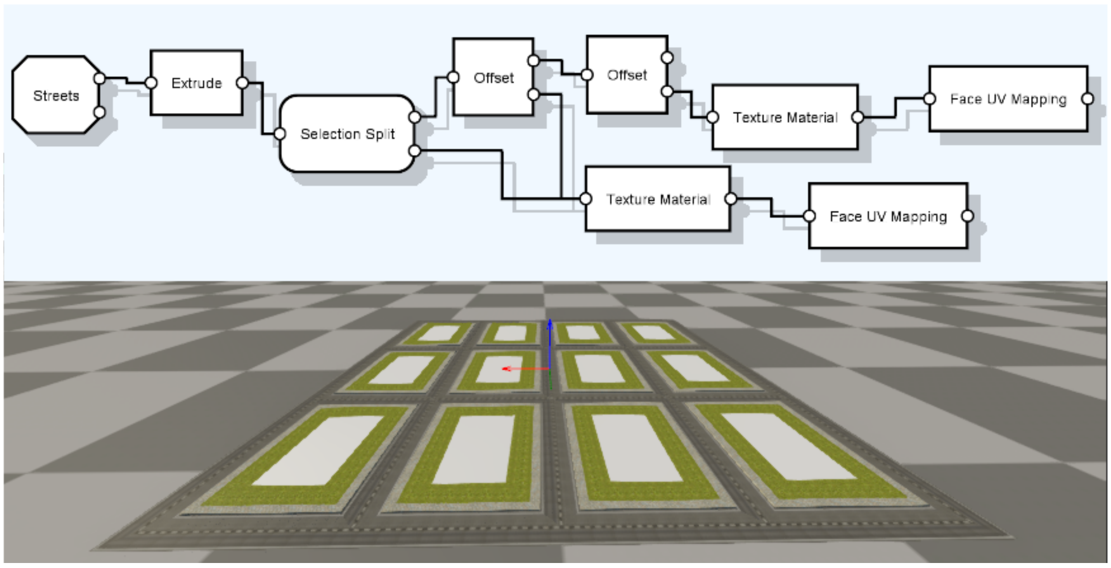


Figure 8.22: Sidewalks and gardens on the built lots.

- First Texture Material (Grass section): Texture="Assets/Ground/Grass01.jpg"
- First Face UV Mapping: UV=2,2
- Second Texture Material: Texture="Assets/Ground/CobbleFloor01.jpg"
- Second Face UV Mapping: UV=2,2

Almost done! Drag and drop the house node into this graph. As you see, it has one input port (which we explicitly declared) and one output port. We didn't specify any gates, meaning that the node should not have any output ports. But since we defined that option "Aggregative", all the data is aggregated into one output. This saves us a lot of work - otherwise we would have to mark all outputs as gates...

Anyway, connect the house to the empty port of the offset node. Let's try to execute it now.

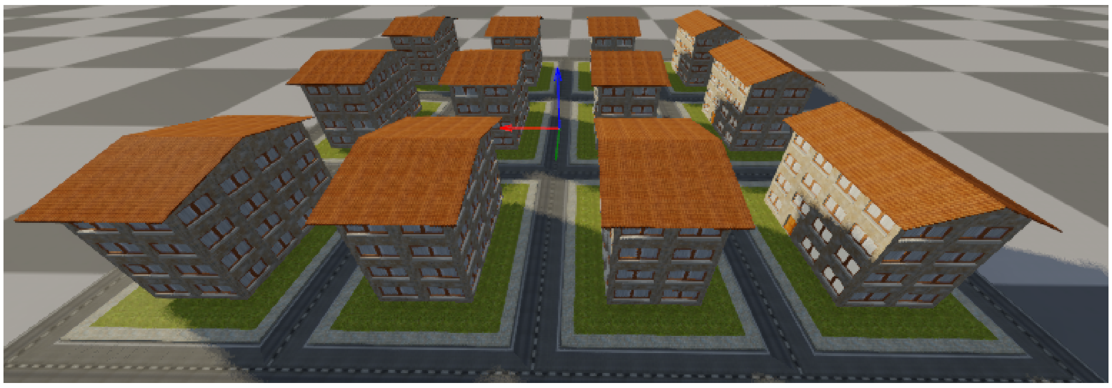


Figure 8.23: Generation of buildings on the lots.

Cool, but all the houses look the same. Let's add some randomness. Go to the graph properties and create a new global parameter of type "Random" and just call it "Random" as well. Don't worry about the initial value.

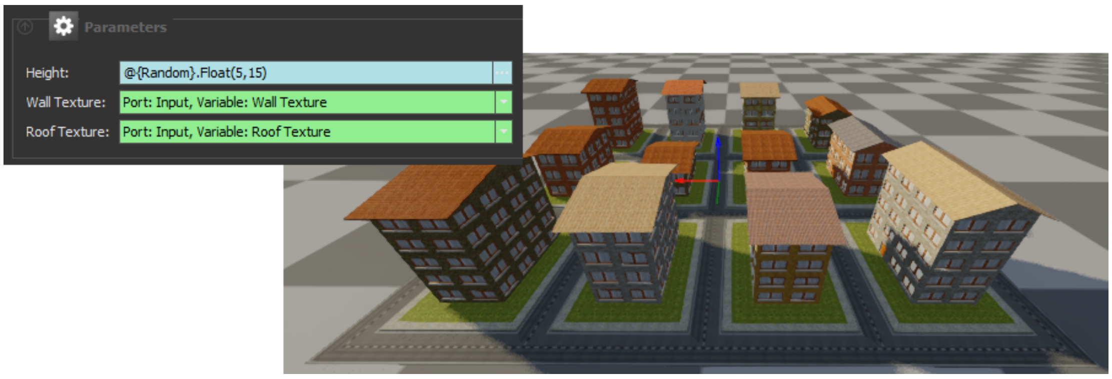
Now, add a node called "Random File From Folder" and put it between the offset and the house node. Select the folder with the roof textures ("Assets/Roofs") and create a new attribute using the "+" (just call it "Roof Texture"). What this node will do is go that texture folder and put the path of a random file in that folder in that attribute. We'll use it soon.

Add a second node called "Random File From Folder" and put it between the previous "Random File..." and the house node. Select the folder with the wall textures ("Assets/Wall") and create a new attribute using the "+" (just call it "Wall Texture").

If you followed all the instructions until now, your "House" node should have at least the parameters "Height", "Wall Texture" and "Roof Texture". Set the mode of the "Height" parameter to "Expression" and type in the following expression:

@Random.Float(5,15)

This will lead to a selection of a random value between 5 and 15 for the height of the building! For the other two parameters, choose "Port Variable" as the mode. Here, you can choose to use the attribute names that were randomly selected by the "Random File From Folder" nodes.



Height: @{Random}.Float(5,15)
Wall Texture: Port: Input, Variable: Wall Texture
Roof Texture: Port: Input, Variable: Roof Texture

Figure 8.24: Neighborhood with varied building characteristics.

Now this looks a bit more interesting! Do you think you can make it better?

Already done?

Here are some suggestions for further work, if you're interested:

- Have you tried playing around with node parameters? Experimenting different combinations and see its effects?
- Add more randomness! Random roof sizes, random garden textures, etc.
- What about trying to add a chimney to the roof or a nice fence around the buildings

gardens? Some splits, extrudes and offsets would be useful in the process...

- Try introducing a new style of window and mix it with the original one within the same building (hint: use the conditional node with some more complex mathematical expressions).

Survey

Would you be so kind so as to answer a quick survey online (8 questions)? The survey is available at:

`https://www.surveymonkey.com/s/X2WMTXJ`