Masaryk University
Faculty of Informatics

Procedural Generation of Spruce Trees

Bachelor's Thesis

Tomáš Stolárik

Brno, Spring 2017

Masaryk University
Faculty of Informatics

Procedural Generation of Spruce Trees

Bachelor's Thesis

Tomáš Stolárik

Brno, Spring 2017

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Tomáš Stolárik

Advisor: doc. Fotios Liarokapis, PhD

Acknowledgement

This is the acknowledgement for my thesis, which can span multiple paragraphs.

Abstract

This is the abstract of my thesis, which can span multiple paragraphs.

Keywords

keyword1, keyword2, . . .

Contents

1 Introduction

1.1 Procedural trees

Procedural generation (also called random generation) is a method of algorithmic creation of data to avoid doing it manually. It has two main uses: computer graphics and video games. In graphics, a common use is creating 3D models and textures. In games, it is used to create large amounts of content in a game automatically. There are many advantages of procedural generation in games, for example, smaller file sizes, ability to create larger amounts of content in a short time, and randomness, which results in different gameplay experience every time.

Procedural generation of 3D models is also called procedural modeling. It covers many different techniques in computer graphics for creating models (and sometimes also textures for these models) from sets of rules. The rules are either set in the algorithm or chosen as parameters. The output is called procedural content and it can be used in computer games, films and in other areas and the user may also further edit the content manually. Procedural modeling is often applied when it would be too difficult to create a 3D model using 3D modelers, or when more specialized tools are required. This is often the case with plants, architecture or landscapes. The most common of procedural modeling techniques are L-Systems, fractals, and generative modeling.

Fractals use one rule repeatedly until a chosen amount of iterations is reached. With this method we achieve the so-called self-similarity trait, which means that parts of the object have the same shape as the object as a whole; they are only smaller. Many real life objects have this trait or are close to it which makes fractals the ideal technique for modeling them. Examples are river networks, lightning bolts and simple plants, such as algae. Fractals are sometimes used for more complex plants like trees, but the disadvantage of a fractal tree is that it looks too regular and unnatural because of repeating the same pattern over and over. Generative modeling uses the programming language called GML It is most commonly applied in modeling and designing man-made objects such as buildings and furniture, as opposed to fractals and L-systems which are mainly used for natural objects.

L-systems are specifically designed to mathematically define plants for 3D model generation. They are special kind of grammars that replace each symbol every iteration instead of replacing just the first symbol. They can be parametrized, so variables like width and length can differ for different parts of the objects. They can also employ randomness – for example, there can be multiple rules for one symbol and one of them is picked at random each iteration. Because of these factors, L-systems do not suffer from the same problems as fractals do when modeling trees.

Thanks to this, L-systems are the most common method for procedural tree generation. In order apply this method, there are two steps: firstly, developing L-system rules which define the tree and secondly, creating an interpreter that generates a model based on the L-system output. An existing interpreter may also be used. There are many L-system rules available for various trees, although not all of them are for a specific type of tree. There are also interpreters in different programming languages or engines, but most of them are only in 2D and many of them do not support all functionalities needed for a complex tree.

1.2 Aims of thesis

In my thesis, I create a procedural tree generator using L-systems. I focus on a specific type of tree – Norway spruce. I develop my own L-system rules because efficient rules are lacking for this type of tree, but I base it on an example L-system (which is not for this specific type of tree, but I am modifying it to fit Norway spruce) from the book Algorithmic Beauty of Plants. I also implement my own L-system interpreter as a C# script in Unity, and I do not build it from scratch either but I use parts of existing code, modify it to fit my needs and add desired functionality such as parameters and randomness.

My goal is for this generator to produce as realistic Norway spruces as possible. Depending on the level of realism achieved, it can be used either for virtual reality applications or computer games. An important factor is for each tree to be a little different from the others, so when multiple trees (or an entire forest) is needed for an application they are not all identical, as it would feel very unnatural. I want to achieve this by adding a suitable amount of randomness to the tree definition.

1.3 Thesis structure

2 Background

2.1 Procedural generation

The term Procedural generation is closely tied to two fields: video games and computer graphics (and very often to both). Some sources refer to these uses of the term as two different things [1], calling them procedural content generation (PCG) and procedural modeling respectively. Others, including this paper, understand procedural generation as a single method further differentiated by the type of generated content.

The term procedural refers to the process that computes a function. [1] Procedural generation is defined as the creation of content automatically using algorithms. [2] This content does not necessarily need to be for games, although that is the most common and best known use of this method. Other applications include for example virtual reality.

The content created using this method is called procedural content. If games are used as an example, almost anything in the game can be procedurally generated: levels, maps, game rules, textures, stories, items, quests, music, weapons, vehicles, characters, etc. [3] This paper focuses on procedurally generating 3D models (more information in the next section Procedural modeling) but briefly mentions other types of procedural content in this section to better explain procedural generation in general.

Procedural generation has been used many times for its many advantages (discussed further in this section). Some famous examples include dungeon generation in Rogue (AI Design 1980) and Diablo (Blizzard 1996), map generation in Civilization (MicroProse 1991), weapon generation in Borderlands (Gearbox 2009) and vegetation generation in SpeedTree (Interactive Data Visualization 2003). [2] A recent well known example is the infamous No Man's Sky (Hello Games 2016) with entire procedurally generated universe.

A good way to distinguish what procedural generation is might be to first define what it is not. Many games contain some sort of content editor – either map, level or character editors. A good example of this is Spore (Electronic Arts 2009) which features creature creator.

In other games, such as Sim City (Maxis 1990) or Minecraft (Markus Persson 2010) the creation (building) of content is a central mechanic of the game. However, these are not considered procedural generation, because the creation of the content relies heavily on user input. [2] There is, of course, no rule saying that procedural generation can include no user input, but if there is some, it should be limited or indirect. [3]

Procedural generation is also called random generation. This means the content generators often include some stochasticity – there are constraints to what can and cannot be generated, but within these constraints the results can vary according to a pseudo-random process. However, this is not a rule and procedural content can also be deterministic. [2]

One of the most obvious reasons to generate content is removing the need to have a human designer or artist create it manually, as human work is expensive and slow. In the past, procedural generation was usually chosen to lower the file sizes because it removed the need to store the content. Other reasons to choose generating content procedurally over manual creation are following: some content is too difficult or time-consuming to create manually while algorithm to create can be rather simple and previously mentioned stochasticity can bring much more diversity than manually created content. This advantages assume the creation of content that was previously made manually but procedural generation also opens new possibilities that were impossible with manual creation. For example, if a software can create content at the same (or higher) speed as the user "consumes" it, the result is for the user the content never ends. [3] Another interesting reason to use procedural generation is to help us understand the design of a content item. When humans design content, they might not fully understand the process themselves (the "whys" of the design decisions) but when they try to implement a program to do it for them, they are forced to think about it, which might provide interesting results (possibly different from the original).

The implementations of PG methods can be referred to as solutions to content generation problems. These solutions might have different desired properties, depending on the problem being solved. Common properties include: [3]

- Speed
- Reliability
- Controllability
- Expressivity and diversity
- Creativity and believability

2.2 Procedural modeling

Procedural modeling is a type of procedural generation with 3D model as the output content. Procedural modeling techniques are successfully employed in many domains, including urban planning, computer games and movie production. [4] One of the aims of computer graphics for a long time has been to proceduralize models. As applications grow in complexity and content, it becomes more difficult for artists to provide the necessary amount and detail of models. In the past, the focus of procedural modeling was only on automating models of natural phenomena (clouds, fire, water, plants) and only recently the community started focusing also on manmade objects. [5]

Current trends in the gaming industry are that gaming companies are under pressure to provide more and more content (which includes mainly models) in their products and this forces them to hire increasing number of artists. Making use of procedural models would be a great advantage for gaming companies if used correctly as it would lower the costs used on artists. [5] Despite extensive work in geometric modeling for over four decades, it remains a time-consuming task and learning how to use geometric modeling tools can require significant training. Many objects and environments contain repetitive and self-similar structures which can be modeled more easily using procedural modeling techniques. These techniques are designed to automatically or semi-automatically generate complex models. They include techniques based on shape grammars, scripting languages, L-systems, fractals, solid texturing, etc. These approaches can be used to generate many complex shapes, but each of them is designed for a specific class of models and requires considerable user input if used for different class. [6]

One of the biggest advantages of procedural modeling is its data amplification capabilities. This means that a simple set of input parameters and/or a few generation rules can produce a wide variety of models. Another important property of PM is data compression. A rather complex geometric model can be represented as a compact procedural model with a set of parameters and the geometry is generated only when needed. This is even more relevant because of current state of computer hardware, because more functionality is currently shifted to GPU's shaders, so it is viable to run instructions to generate data instead of storing it. [7]

Although PM promises high productivity gain, compact representation and a seemingly endless variation, most of its current methods are still not a suitable alternative to manual modeling due to the procedural models' poor controllability. Users are required to manipulate complicated rules and/or parameters without direct control over the output which makes them unable to predict it. [7] Different procedural modeling techniques require varying amounts of user input. Using high amount of user input has both advantages and disadvantages. Without sufficient user input, the generated result might be too random and different from user's intent, while too much user input can be time-consuming and overwhelming for the user. Ideal setting is, when user can provide any amount of input he chooses and the algorithm adjusts accordingly. [6]

Because of its benefits, one of the most fitting uses for PM is creating virtual worlds for games and simulations. Most of these kinds of applications require the virtual world to be as extensive and detailed as possible, which greatly increases the time and money investment for manual modeling. Therefore, a consensus is growing that the solutions to this problem will be procedural modeling techniques. [7]

2.3 L-Systems

2.4 Procedural trees

3 System architecture

4 Methodology

5 Implementation

A 3D application was developed as a practical part of this thesis. This chapter describes its development.

5.1 Symbol replacement function

L-systems were chosen as the procedural generation method for this application because of their effectiveness in modeling plants as it was their original intended use. Because of this, early phases of the application development were focused on trying out examples from Algorithmic Beauty of Plants (a comprehensive guide to using L-systems for procedural modeling of plants).

First task in development was to create a function which gets an input L-system grammar rules, number of iterations and an axiom and outputs the resulting word. This is essentially a simple replacing algorithm used for computing grammar's words, the only modification to fit L-systems was to replace every symbol which has a fitting rule, instead of just the first one.

For testing the function, an example L-System grammar from Algorithmic Beauty of Plants was used, so its correctness could be easily determined by comparing with the book's result.

5.2 Modeling based on the grammar output

Next, there was a need for another function to create a 3D model based on the grammar's output. The symbols and their meaning for modeling were chosen as follows. Because the prototype in this phase was only drawn in 2D plane, the basic turtle graphics symbols F, +, -, [, and] for moving in two axes were sufficient with their meanings being to move forward, rotate clockwise, rotate counter-clockwise, save state and load state respectively.

The State class was implemented for storing information about position and rotation while drawing. The current state was modified by move forward and rotate operations and save and load operations pushed to and popped from a stack of states. The move forward operation, besides modifying the position, also had to implement the actual drawing by connecting the previous and the next position by a line.

The first implementation of this function drew inspiration from a script available online and also needed information from the documentation of the Unity Mesh class. For simplicity, the mesh was just a block of constant width and length (as previously mentioned, the drawing was using only 2D plane at this point although it was implemented in Unity's 3D scene). The first four vertices of the block were defined by the position before moving and the next four by the position after moving. If the rotation was not zero, the new position was determined by basic trigonometric functions. The length of the block and rotation were parameters of the script and were the same for every movement/rotation operation.

For testing the implementation of the function, again a sample L-system grammar from ABOP was used. To view it from different sides in the Play mode, the Unity standard asset FPS controller was added to the scene along with some floor.

5.3 Addition of rotation in all axes

To make the transition from a flat model with no movement in Z-axis to a fully 3D model, symbols for rotation in other axes had to be added to the grammar's alphabet. For this, following symbols from extended turtle graphics were used: ^, &, \and /. The State class was extended to contain three rotation angles (pitch, yaw and roll) instead of a single one. The rotations were first implemented using trigonometric functions. This implementation was needlessly complicated and the result was also incorrect. The main reason was that rolling rotation could not be correctly implemented using one position and three rotation angles because it does not change the direction of the immediately following segment, it only affects further rotation and the current representation of the state could not define that.

An L-system grammar representing an actual tree was used for testing this time. This L-system was parametric and parameters for the rules were not supported at that time in the application, so a modified version was used with the parameters left out. The result was much different from the expected due to the following reasons. The first was the incorrect implementation of the rolling rotation and the second, there were no parameters and no randomness, therefore every branch had the same length, width and rotation.

5.4 Parametrization of the L-system

The L-system needed to support parameters in the rules to be able to have different lengths, widths and rotation angles for individual branches. The parameters and their values were stored in a dictionary and another replacing function was created and included in the original function, that replaced formal parameters with their values. The drawing function had to be reworked to draw based on the parameters that immediately followed the symbol representing an operation. The move forward operation had parametrized length while rotate operation had parametrized angle and a new operation was introduced - change width, with width as a parameter and ! as a symbol. The non-terminal symbols had variable number of parameters.

A random multiplier was added to the lengths, widths and angles in the draw function to promote variability, because at the time, the tree looked very regular and unnatural.

5.5 Correction of the rotations

A simpler implementation was found for the rotation, which could be more easily corrected. The representation of the state was switched from 1 position and 3 rotation angles to one position and one direction. Instead of calculating the next position from rotation angles using trigonometric functions it is just using the heading direction, which is updated after every time rotation symbol is read. The rotation update works using quaternions, which have built-in support in Unity and therefore it is relatively simple to implement. This representation of the state had the same problem as the previous one - incorrect rolling rotation, due to not having any immediate effect on the direction nor position - but the calculation of the next position was significantly simplified (it consisted only of multiplication of the direction vector by length and adding it to the current position), so it was closer to the solution.

The solution was to use 3 direction vectors - up, left and heading, instead of just one. The vectors are initialized to the coordinate axes and every rotation, two of the vectors are rotated around the third. This way even the rotation that does not affect the immediate heading rotation (roll) works correctly, because it changes the other vectors, that will be used in further rotations. For implementation, quaternion rotation of vector around other vector was used. Testing determined the rotations to be finally correct.

5.6 Adjustment to a specific tree

Spruce tree was chosen as the type of tree to be generated. The L-system was changed slightly to roughly match the shape of this type of tree according to photos on the internet. Mainly the angles and lengths were changed. An observation was made that the angle of the main branches of the spruce tree change depending on their height - lower branches have much bigger angles than the upper ones. Because of this, a multiplier was added to the angle parameter in the L-system that describes this.

More detail was added to the model, switching from four-sided branches to eight-sided. As a more specific type of tree, Norway spruce was chosen. The reference images from the internet were not sufficient for things like the branching structured, as much more detail was needed. Norway spruces are relatively common in Czech republic and Slovakia, which made them a good choice, because it was possible to take photos of them without traveling. Those photos had more detail and were therefore suitable as reference images for modeling. The branching structure in the grammar was modified to match the reference photos.

5.7 Addition of leaves - first version

First version of the leaves (or needles) was defined the same way as the branches - a hexagonal prism, the only difference was they were smaller and green. This was not a good way to do it, because it made the leaves eight-sided, which was unnecessarily complex for such a small object. It was not even possible to see how detailed they were, because they were very small and also, real spruce's leaves are only four-sided, so they were actually more round than the real version. The leaves were also added to the definition of the tree the same way as the branches, which means there was a new symbol added to grammar's alphabet that represented a leaf of given length and width (defined by its parameters) and it was used at appropriate places in the grammar's rules.

At the time, the tree had a fairly complex branching structure, especially the lower branches had too many subbranches, some of them too small to notice. That means when leaves were added in the grammar definition to every branch of any size, it turned out to be an extremely high amount of leaves, which were quite complex. This resulted in the application being very difficult for computer performance, due to having about 4 million polygons. Because of this, the application ran at about 3 frames per second.

This made the application very difficult to work with, so these changes were reverted and the focus of the work was switched to other things until a better solution for the leaves is found. The leaves defined in the L-system also made the grammar more complicated and less readable, so the next version of the leaves would be implemented differently.

6 Results

A An appendix

Here you can insert the appendices of your thesis.