# Procedural tree generation

## Introduction

### 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.

### 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.

### Thesis structure

## Background

### 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

### 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]

### L-Systems

### Procedural trees

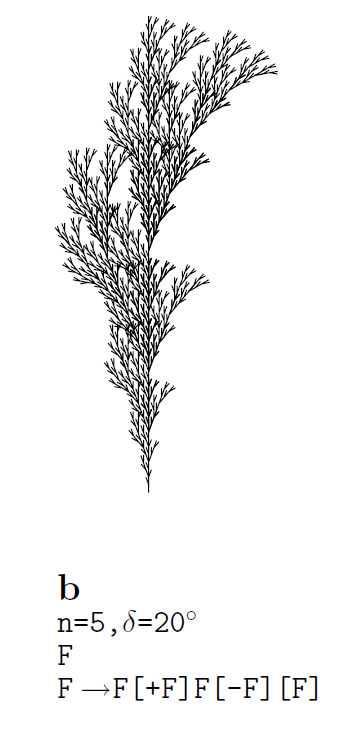
## System architecture

## Methodology

## Implementation

I started by reading Algorithmic beauty of plants (first two chapters). When I had read most of this I started implementing, because I already had most of the necessary information.

I defined length, width, branching angle and number of iterations as public attributes so I could modify them from Unity inspector. I also added rules as a dictionary from char to string. First thing I needed was a function to replace symbols each iteration. I used the Replace() function from here: <https://gist.github.com/michidk/ced97c81388fa8a0d6d9>.

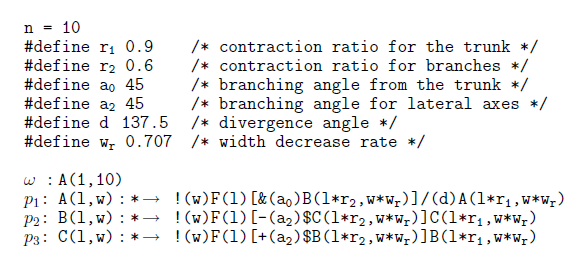
Then I wanted to do an L-system interpreter, which draws according to computed symbols. I first did it only in 2D (it was actually 3D, but I didn’t do any rotations in Z-axis, so the image was visible only from one side) so I just needed to implement basic turtle graphics symbols – F,+,-,[ and ]. I used the State class from zhere: <https://gist.github.com/michidk/ced97c81388fa8a0d6d9> to remember the position and angle, which were modified by F, + and -. For symbols [ and ] that represent the saving and loading of states I used a Stack of states. Then I needed the actual creating of the mesh using F symbol. I used part of this script: <https://forum.unity3d.com/threads/l-systems-for-unity-free-script-included.272416/> . For understanding Unity mesh class I needed this: <https://docs.unity3d.com/Manual/GeneratingMeshGeometryProcedurally.html> .For simplicity the mesh was just a cube of a given length and width. First four vertices were defined by a position before moving forward with F symbol and second four vertices were defined by position after moving. If the rotation was not zero I determined the new position using goniometric functions. Because I just wanted to test this I used a sample L-system from the book, that I knew how it should look like, so I would know if my implementation is correct.

< - I used this L-system and the output looked correct. I also put in Unity FPS controller so you could move around the scene.

The FPS controller wasn’t ideal for this, so my next step was to replace it. I used this script from the Asset store: <https://www.assetstore.unity3d.com/en/#!/content/61537> . I also wanted to make an actual 3D tree, so I needed to rotate in all axes. I added new symbols for other rotations: ^, &, / and \. I put three rotations into State (pitch, yaw and roll) for all three axes.

I implemented all three rotations using goniometric functions, which was a) too complicated and b) result was incorrect. The reason was that rolling rotation couldn’t be implemented in this representation because it doesn’t change the direction of the next branch, it only affects further rotation and my current representation (1 position + 3 angles) couldn’t define that.

I chose the L-system below to check my implementation of 3D L-system. Because I didn’t have support for parametric L-system yet I left out all the parameters from it. The resulting tree looked very weird and very different from the drawing in the book because of multiple reasons.

First – incorrect implementation of rolling rotation, second – no parameters and no randomness, therefore every branch had the same length, width and rotation. So my next two steps were to add randomness so not every branch is the same, support for parameters and more importantly correct the rotations. So I added a dictionary of parameters and their values, and then I reworked both the string replacement function and the draw symbol function to work with these parameters. To do so I needed an extra function to replace formal parameter with its value. I added a random modifier to lengths widths and angles when drawing.

I also needed to choose a specific type of tree that I would focus on, so I picked spruce tree. I changed the L-system a bit to roughly match this type of tree according to photos on the internet. I mainly changed the angles and lengths. I noticed that the angle of the main branches of the spruce tree changes depending on their height. Lower branches have much bigger angles than the upper ones, so I added a modifier to the L-system definition that describes this.

I also added some more detail to the model, switching from four-sided branches to eight-sided. Only after adding all these changes did it occur to me that the rotations can probably have simpler implementation than mine and I could probably correct a simple implementation more easily. So I switched from one position and 3 float rotations and goniometric functions to calculate the result position to one position and one direction, which was rotated using Quaternions every time a rotation symbol was read. This representation had the same problem as the previous one – incorrect roll because it had no immediate effect on the direction nor position – but it was simpler and the calculation of the next position was much simpler (just multiply direction vector by length and add it to current position), so I was getting close to the solution.

The solution was to use 3 direction vectors – up, left and heading. It was also mentioned in the book but I didn’t notice it at first, I had to figure it out myself. The vectors are initialized to the coordinate axes and every rotation, two of the vectors are rotated around the third. That way even the rotation that don’t affect the immediate heading rotation (roll) work correctly, because they change the other vectors, that will be used in further rotations. For implementation I used Quaternion rotation of vector around other vector. My rotations were finally correct.

Next I had to tune some parameters so they fit the type of tree I selected. For example, the branches were too thick compared to the trunk. I also had to tweak the randomness a little, because I was sometimes getting too big angles, too long branches etc.

I also tried assigning a texture to the trunk but I made a mistake defining uvs, so it didn’t work.

I decided that I will be doing Norway spruce specifically. I didn’t have very good reference images for things like branching structure. I only had the ones from the internet, which weren’t detailed enough. So I went out and took some photos for reference. Norway spruces are quite common in Czech republic, so that was also a reason to choose this tree.

I modified the branching structure in the grammar to match the reference photos. Next I tried to work on the leaves. I defined them the same as branches – a hexagonal prism, the only difference was that they were smaller and green. This wasn’t a very good idea, because it made the leaves eight-sided which was unnecessarily complex for such a small object (you couldn’t even see how detailed they were, because they were very small, and also real spruce are only four-sided, therefore I made them more round than they really are). At the time I also had a fairly complex branching structure – especially lower branches had far too many subbranches (some of them almost too small to see). So when I added leaves to grammar definition I got some leaves on every branch, no matter how small, which turned out to be an extremely high amount of leaves, which were quite complex. This resulted in application being very difficult for computer performance (4M tris – cca 3 fps).

It turned out to be impossible to work with the application with such a bad performance so I took a step back and removed the leaves, planning to focus on something else and returning to them later. I still needed to do some optimalizations, because even without the leaves, my tree had quite a lot of tris (much more than necessary for this amount of detail). One optimalization I found was removing the top and bottom faces of the segments on non-terminal branches which were completely unnecessary because you couldn’t see them.

I also realized that my tree was far too detailed on some places (lower branches having many subbranches) while being too simple from the distance (too few big branches). So I modified the grammar to create two levels of big branches per iteration instead of just one and also decreased the amount of created small branches per iteration by half. The resulting tree looked much better (much more dense) while also having less tris – better performance. So this turned out to be an optimalization as well as visual upgrade.

Next I wanted to upgrade from straight line branches to curves. I used this tutorial: <http://www.wasabimole.com/procedural-tree/how-to-generate-a-procedural-tree-in-unity-3d/> , more specifically the idea of applying a random rotation, so the branches get twisted. I used it in a function that creates a segment by randomly rotating the heading direction. So the result wasn’t really a curve because of 1) long straight segments (but much shorter than before) and 2) randomness. I had one other problem and that was determining how big can the maximum angle for curving be. When it was too small it didn’t make much difference because the branches still looked more or less straight. When it was too big, branches looked better, but the trunk was curved which looked very weird. I thought about making separate code for curving branches and for trunk but then I found a better solution. Essentially this was reverse proportionality between curving angle and width of the segment (trunk is thick, so it is only curved very little, then the smaller the branches are, the more curved they can be). This worked pretty well with trunk being almost straight and branches being differently curved.

I also discovered a solution to my previous problem with texture. I put a texture I cropped from a photo I took, but the resulting model was all one color. This was because I assigned wrong uvs – I thought I put them as coordinates of the picture (pixel number) but instead they should be in range between 0 and 1. I corrected this and the texture appeared on the model. However, this uncovered another problem – the textures on neighbouring segments didn’t align properly.

Current grammar

A(l,w,a0) -> !(w)F(l\*r2)[E(l,w\*wb)]\(d2)[E(l,w\*wb)]\(d2)[E(l,w\*wb)]\(d2)[E(l,w\*wb)]/(d)F(l\*r2)[E(l\*r5,w\*wb)]\(d2)[E(l\*r5,w\*wb)]\(d2)[E(l\*r5,w\*wb)]\(d2)[E(l\*r5,w\*wb)]/(d)A(l\*r1,w\*wr2,a0\*ar)

B(l,w) -> !(w)F(l\*r2)[-(a2)&(a2)G(l\*r3,w\*r2)]!(w\*wr2)F(l\*r2)[-(a2)&(a2)C(l\*r2,w\*wr)][+(a2)&(a2)C(l\*r2,w\*wr)]B(l\*r1,w\*wr)

C(l,w) -> !(w)F(l\*r2)[+(a2)&(a2)G(l\*r3,w\*r2)]F(l\*r2)[-(a2)&(a2)C(l\*r2,w\*wr)][+(a2)&(a2)C(l\*r2,w\*wr)]C(l\*r1,w\*wr)

D(l,w) -> !(w)F(l)A(l,w\*wr,a0)

E(l,w) -> !(w)&(a0)F(l\*r4)B(l\*r2,w)

G(l,w) -> !(w)F(l\*r2)[-(a2)&(a2)G(l\*r2,w\*wr)]G(l\*r1,w\*wr)

N(l.w) -> f(l\*lr)M(l\*lr)f(l\*lr)M(l\*lr)f(l\*lr)M(l\*lr)f(l\*lr)M(l\*lr)f(l\*lr)M(l\*lr)f(l\*lr)M(l\*lr)

M(l.w) -> [+(d2)&(d2)L(l\*lr)][-(d2)&(d2)L(l\*lr)][-(d2)^(d2)L(l\*lr)][-(d2)&(d2)L(l\*lr)]\(a2)[+(d2)&(d2)L(l\*lr)][-(d2)&(d2)L(l\*lr)][-(d2)^(d2)L(l\*lr)][-(d2)&(d2)L(l\*lr)]\(a2)[+(d2)&(d2)L(l\*lr)][-(d2)&(d2)L(l\*lr)][-(d2)^(d2)L(l\*lr)][-(d2)&(d2)L(l\*lr)]

r1: 0.9

r2: 0.5

a0: 110

a2: 38

d: 137

wr: 0.707

d2: 90

ar: 0.9

wb: 0.1

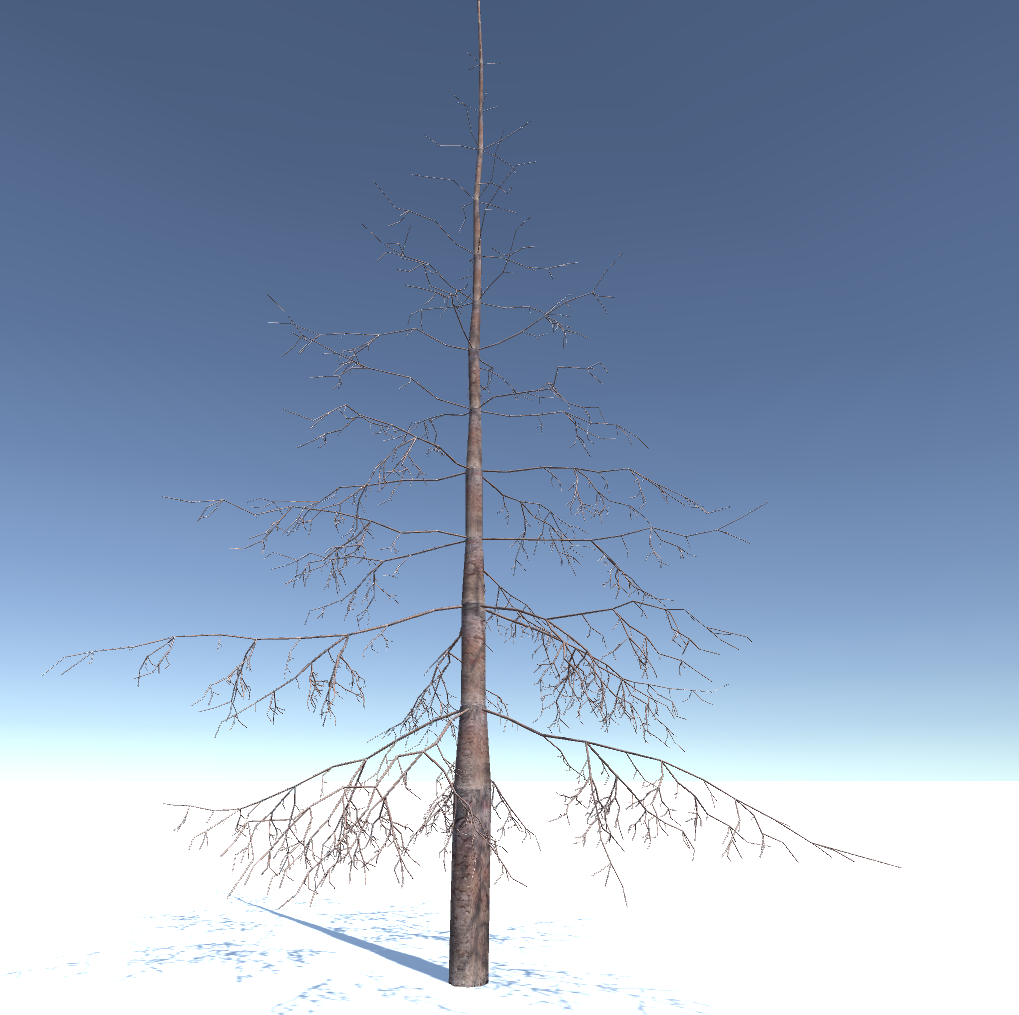
r3: 0.4

r4: 0.25

lr: 0.15

r5: 0.95

wr2: 0.707



^ a close-up on the segments that don’t align properly

So my next task was to correct this. First I made the texture seamless in Adobe Photoshop using this tutorial: <https://www.youtube.com/watch?v=chFSIUXe1Ro>

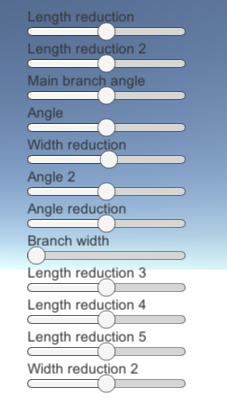
This helped for texture repeating on the same segment but wasn’t enough for textures on the neighbouring segments because the vertices for the top of one segment weren’t at the same position as the bottom of the next segment. This was because rotations were applied after drawing the first segment and before drawing the second one, so the segments were rotated from each other. My implementation was following: I had a current state (position, left, up and heading vector and length), I applied rotations to it, created bottom side of the segment from it, then made a new state named nextState, which was the same as current except the position that was calculated current position + heading vector \* length of the segment. This worked well except for the straight following segments that were rotated from each other, that had the aforementioned problems.

What I actually needed was to have already two states – current and next and apply all rotations and changes of width to the next one instead of current one. Then I would create a segment between the current position and the next one and the rotations would happen on the segment instead of between them, so I wouldn’t get this bad fitting of following segments. When I first implemented this, I had many problems, because I didn’t realize my stack of states didn’t support having two states at all times (I always pushed/pulled only current state) so when I used push and then pop, the rotations were reset, which wasn’t desired. So I changed the stack to always push and pop both the current and next state.

After this change, the vertices and the texture aligned properly. After that I had two problems: the trunk was strangely rolled and the main branches were very thin at the start (but normal width after). I first tried to find mistakes in the code, but I realized it could be fixed easily by putting the whole part of the grammar that forms one level of branches into brackets – saving the state before creating the level and restoring it after.

Next I started to work on the interface. I added a slider UI element into the scene for every parameter of my grammar and an update button. Every slider had the same name as the parameter it modifies and I use this fact in the redraw function. Redraw function is used when you click on the update button and it puts all the slider values into the corresponding parameters based on the names, destroys the current tree, recalculates a new one with the new values and draws it. I set the default slider values to the default values I wanted to have for the parameters and I also set some reasonable minimum and maximum values for each slider. Then I added labels for sliders, so the user knows what parameter he is editing.

Next, I had another try at the leaves. The previous time I had big problems with the performance (the leaves were as detailed as the branches – too much detail) so I knew I had to make them simpler this time. I chose a pyramid shape, because it has only three sides (I don’t need the bottom) and still looks decent (it is at least 3D). I create a function that builds a leaf and I wanted to do their placement the same way as I had before: a letter in a grammar that translates into “create a leaf here”. I had lot of problems with this approach, mainly because I had no “synchronization” between the leaves and the branch it should be attached to – sometimes leaves didn’t cover the whole branch and sometimes leaves went off the branch. I used a lot of *f* symbols which made the state move without drawing to make space between individual leaves and it was very difficult to match them with the branch length. I also had another idea how to make leaves work in mind so I tried it, because the previous one didn’t work very well.

The idea was to not define them in the grammar (as this was very inconvenient for previously mentioned reasons) but instead create them while creating the branch. I didn’t want to create leaves for every segment (for example I didn’t want leaves on the trunk) so I chose it would be decided whether the segment will have leaves or not based on its width. I set a threshold for the segment width and only when it is lower than the threshold, the leaves are created. The leaves creation itself works in groups, first a point on the branch is selected and then a group of leaves is created, all of them starting on the same point but rotated in a different direction. Then another point is selected, by moving from the first point along the branch by a selected offset. On this point, another group of branches is created. This continues until the end of the branch is reached or until a certain amount of leaves is created (I wanted to limit the number of leaves to avoid performance problems).

Next I added a texture to the ground (I found it on the internet). It was stretched so I changed the tiling attribute. I recieved feedback to the leaves I made and it was that they look good enough from the distance, but it would be better if at close range they had the correct (four-sided shape) and also that there is not enough of them. I chose both the simpler shape and the limited amount specifically to have acceptable performance so I realized I had to implement level of detail – something that would have helped me sooner if I added it.

I created 3 different generators for 3 LODs, each creating one tree. I chose different amounts of iterations for them to vary in complexity of the grammar output. Then I created new C# script for LOD that calculates the distance between the camera and the tree and chooses the corresponding level of detail – enables the right model and disables the others (Unity setEnabled function). Then I created alternative geometry for leaves for the highest level of detail that is much more realistic (four-sided). I also added separate constants maximum/minimum offset between leaves and maximum number of leaves per branch for each level of detail. Offsets are higher for low LODs and vice versa and number of leaves is low for low LODs and high for high ones. I added a reset scene button and text describing controls on the UI.

This solution for LOD had some problems: 3 levels weren’t enough because in the future there will be many trees in the scene, so the detail should scale to very low and more importantly since I was basing it on iterations, the shape of each tree was different and switching between LODs was very noticeable. So I added more LODs (I tried different amounts then I chose 6) and reworked the generating of different LOD tres. I wanted to basically the same tree 6 times, but with each lower LOD some small branches would be skipped. This way the poly count would be lower but the overall shape of the tree would be the same (only with different leaves still). To do this I first needed to change the order of generation – because of randomness in each branch generation, I could no longer generate one LOD tree after another if I wanted to keed the shape the same, because I couldn’t produce the same randomness. Instead I started generating the trees in parallel – the first branch of every tree is generated, then the second of every tree etc. Then I determined thresholds for every LOD – if the branch being generated would be thinner than the threshold it is skipped and therefore less branches are generated for lower LODs. Another way to differentiate LOD versions‘ poly count while keeping them visually very similar was to create different poly counts for individual branches – make branches with different numbers of sides (I chose 4, 6 and 8). For that I needed to implement methods that calculate uvs and tris for corresponding number of sides and modify the method for calculating vertices to take number of sides into account. The resulting LOD chooses 8- and 6-sided branches respectively for two highest LODs and 4-sided for the others. As an optimization, I also use 4-sided branches for smaller branches even on high LODs.

# Results

## Bibliography

[1] <https://en.wikipedia.org/wiki/Procedural_generation>

[2] <http://julian.togelius.com/Togelius2011What.pdf>

[3] <http://pcgbook.com/wp-content/uploads/chapter01.pdf>

[4] <http://research.michael-schwarz.com/publ/files/cgapp-sig15.pdf>

[5] <https://ccl.northwestern.edu/papers/ProceduralCityMod.pdf>

[6] <http://graphics.stanford.edu/~pmerrell/tvcg.pdf>

[7] http://hpcg.purdue.edu/papers/Smelik14CGF.pdf