Biologically Inspired Large-Scale Simulation of Tree Growth

Master's thesis in Computer science and engineering

RASMUS TOMASSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Biologically Inspired Large-Scale Simulation of Tree Growth

RASMUS TOMASSON

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Biologically Inspired Large-Scale Simulation of Tree Growth
RASMUS TOMASSON

Supervisor: Erik Sintorn, Department of Computer Science and Engineering
Examiner: Mikael Wiberg, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

Biologically Inspired Large-Scale Simluation of Tree Growth
RASMUS TOMASSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis has explored solutions for large scale biologically inspired simulation of tree growth. From previous works a growth engine was adapted and implemented utilizing the parellization of graphics programming in order to simulate and visualize tree growth on a large scale. The implemented algorithm Incorporated three main biological aspects: a light evaluation method, a branch thickness estimate and a energy distribution formula. The implemented method was evaluated in terms of rendering and simulation time and it was measured, on relatively old hardware, to be able to handle simulation of up to eight million branches in a few milliseconds as well as rendering of up to fifty thousand branches within in reasonable frame rates for real time interactive applications. With improvements in terms of performance and rendering, as well as further investigation of biological features to incorporate, the method could arguably be developed into a system capable of simulating and visualizing tree growth on a large scale, with potential use cases in video games and other interactive real time applications

Keywords: Computer science, engineering, project, thesis, Computer Graphics, Visualization, Simulation.

Acknowledgements

Thanks to RapidImages for providing me with technical support and coffee.
Thank you Erik for being such a relaxed and knowledgeable supervisor.
Thank you Levi for being my best friend

Rasmus Tomasson, Gothenburg, June 2022

Contents

1
Introduction

Vegetation is a prominent part in our world and in computerized images and virtual scenes they arguably play an important role in making them look and feel more real. The modeling and rendering of vegetation have applicability in entertainment mediums such as video games and movies, as well as in urban and architectural planning tools among others. Scenes in video games often include rendered vegetation. However vegetation such as plants and trees feature complex and unique surface geometry, unlike for example buildings and structures. Thus creating and rendering the geometry of plants pose a difficult problem. Procedural generation of vegetation concerns the use of algorithms to automatically generate plant data, usually in the form of mesh data. Procedural generation of vegetation is a complex and open problem due to plants' intrinsic geometrical complexity as well as its intricate growth mechanisms. In the case of video games, making realistically looking vegetation previously required thorough work, usually done by 3D-modeling artists. The process of manually designing and creating 3D-models is often time consuming, thus tools for realistic and procedural generation of vegetation have been beneficial for video game development, both economically and artistically. As a result there exist several tools for creating vegetation. For example *Xfrog*[1] which uses a procedural algorithm to create 3D-models of plants and flowers.

Apart from the shape of vegetation being complex and difficult to model, most video games use static 3D-models for plants and trees as opposed to dynamic models that change over time like real plants. The games that do feature plant growth often feature simple and discrete growth processes in which plants only have two or three static model stages. For example in Minecraft[2] every tree in the game only has a sapling state and a full grown state. Furthermore in order to save memory and shorten rendering time, tree models are often duplicated and reused many times in a game, instead of using a unique 3D-model for every tree. Having environments in video games that dynamically change over time in unique and realistic ways would arguably heighten immersion.

In this thesis focus has been on biologically inspired large-scale tree growth. The aim with the thesis was to find solutions for how tree growth could be simulated for many trees and how to render them efficiently. This included several sub problems:

- **How can tree growth be simulated?**
- **How can such a simulation be adapted for large scale?**
- **How can the generated trees be visualized on a large scale?**

In order to save memory storage, tree models are often reused in video games, thus if only unique tree models are to be rendered, their data have to be stored in an efficient way. Storing entire 3D-models of hundreds of thousands of trees would take up an enormous amount of memory. Furthermore the growth algorithm needs to be fast in order to be able to update a large number of trees in a short period of time which was considered necessary for real time applications such as video games. However the intention was not to update trees every frame.

Many projects from previous work either focuses on biologically inspired and realistic looking trees with little focus on performance such as in Pirk et al.[3], or high performance algorithms with specific use cases like for example the simulation of wind movement in trees Pirk et al.[4]. Not much research seems to have been done on large scale simulation of tree growth with biologically inspired growth mechanisms and with a focus on performance. In this thesis such an algorithm was implemented on the GPU utilizing the parallelization of compute shaders to execute the growth process, as well as utilizing GPU-instancing to speedup the rendering process of many trees. The implemented growth algorithm was mainly inspired by the growth algorithm presented in the paper *Self-organizing tree models for image synthesis*[5] and it was adapted to work in a parallell execution process. The rendering times and simulation times of the implemented algorithm was measured, and the algorithm was able to simulate tree growth for up to eight million branches in a matter of milliseconds.

2

Background

2.1 Tropism

Plants continually grow during their lifetime and that growth is a result of either cell growth or cell division[6]. The plant tissue known as *meristem* allows roots and stems to grow and differentiate, and it is found at the tip of roots and buds. Plants usually grow upwards, opposite of the gravitational pull, and in the direction of the highest incoming light. This is called Gravitropism and Phototropism respectively[7]. There are many types of tropisms that affect plant growth and it acts as a directional bias of the growth direction on branches. Tropisms are caused by for example Gravity, Sunlight, Proximity of water among other factors.

2.2 Apical Control

Apical control is the suppression of growth on branches that grow of from the main trunk. These branches are often referred to as lateral branches. Apical control affects the general shape of trees as it determines when and where new branches will emerge. A larger distribution of energy to lateral branches will often result in more shrub like plants with a less dominant main trunk, whereas a larger distribution of energy to main branches will often result in more tree like plants with a distinct main trunk. In trees apical control is imposed by reducing branch photosynthesis and restricting water and nutrient supply. Hormone appear to play in important role [8] in how the mechanisms of apical control works.

2.3 Primary and Secondary Growth

The actual growth of a plant can be divided into two main parts Primary Growth and Secondary Growth[9]. Primary Growth is where the tips of stems and roots grow and become longer, this in turn enables the tree to seek out water or sunlight, by growing in the direction of it. Secondary Growth increases the stem thickness, which in turn results in increased hardiness and strength needed to stabilize the tree as well as increased amount of vascular canals that transport water from the roots to the leafs.

2.4 Related Work

Historically L-systems and fractal patterns have been been used to model the branching pattern of plants. L-systems were presented in 1968 by Aristid Lindemayer[10] and are still commonly used today. They were initially created to describe cell division but has since found applicability in mathematical modelling of complex branching structures. L-systems are similar to a formal language where grammatical rules are applied in steps to a given input. Starting from a given initial state an L-system can generate self-similar patterns making them usable for generating the branching structure of plants.

Figure 2.1: L-System Generated Plants

However the original L-systems were designed without the possibility of communication between the environment and a growing plant, thus making it impossible for growing plants to be affected by a change in environment and for them to affect the environment. Open L-System served as an extension by adding a dedicated parameter dynamically set depending on current environmental conditions[11]. However most recent plant model algorithms seems to favour a more biologically inspired growth algorithm, where the process of gathering light and distributing resources within a tree is simulated.

2.4.1 Biologically Focused Research

In the paper *Synthetic Silviculture: Multi-scale Modeling of Plant Ecosystems*[3] a biologically inspired method for designing large scale ecosystems was presented. Their method adopted a holistic approach to their growth algorithm and included environmental parameters as input such as different types of tropism and competition for resources between plants. Their method supported up to 500k plants to be

simulated, though not in real time. Much like many other methods they make use of GPU-instancing for improving the rendering times of the trees.

In *Capturing and Animating the Morphogenesis of Polygonal Tree Models*[12] a method that is able to compute developmental stages of a given tree model is presented. This in turn is used to animate a growth process that leads to the input mesh. In order to compute the developmental stages from the given model a mesh contraction is performed wherein a tree skeleton is extracted. It is an interesting model that creates intermediate model stages based on a given end stage, however in this thesis the intention was to continuously simulate the tree growth and thus a final end state is not known.

In the paper *Plastic Trees: Interactive Self-Adapting Botanical Tree Models*[13] a technique is presented that allows complex tree models to adapt and interact with the environment. With the technique tree models will react to new physical obstacles inserted into the scene, by regrowing as if the obstacle had always been there. The method could be used both in urban modeling systems but also in video games by minimizing the need for designer to manually alter a tree if the surrounding environment needs to be changed.

In *Interactive Modeling of Virtual Ecosystems*[14] an interactive and biologically based method for modeling virtual ecosystems was presented. It supported up to 140 trees to be simulated in less than two minutes and utilized a space colonization method with a voxel grid for collision detection. Light was evaluated by discretely sampling incoming light during different times, and phototropism was achieved by biasing the growth direction towards the direction of highest incoming light. Tree branches were rendered as cylinder meshes.

2.4.2 Performance focused research

In the paper *Fast Simulation of Realistic Trees*[15] a method for simulating light wind movement in trees using sinusoidal oscillations was presented. The method utilized parallelisation of CPU threading for reaching high performance. Their internal tree models took up about 1 MB of space.

In the paper *GPU-Based Real-Time Procedural Distribution of Vegetation on Large-Scale Virtual Terrains*[16] a framework for real-time procedural distribution of vegetation for large-scale applications was presented. Their approach utilized GPU-parallelisation and GPU-instancing in order to improve performance. They employed a quad-tree in order to partition the terrain and thus manage memory, as well as a hash technique with fixed sized buffers. They minimized data transfers between the CPU working memory and the GPU working memory and handled the distribution of vegetation solely on the GPU.

In *Real-Time Rendering of Plant Leaves*[17] a framework for rendering high fidelity leafs in real time was presented. The framework included a variety of different global illumination effects. Though intended for real time it was not specifically intended for large scale, and due to the leaf meshes high resolution meshes as well as the costly global illumination effect it likely would not fare well for large scale purposes.

In *Interactive Large-Scale Procedural Forest Construction and Visualization Based on Particle Flow Simulation*[18] a method for large-scale procedural generation of forests, and visualization of the forest, using particle flow simulation was presented. Their method enabled visualization of areas with millions of unique trees. The particle based approach starts with particles initialized at leaf positions of a tree. A tree skeleton is then generated by a step-wise simulation of particle movement and by merging particles with their closest neighbours and roots.

3
Theory

Due to the performance requirements of the growth algorithm it may be of interest to use parallel computing when simulating the tree growth. The simulation process for every tree will be rather identical and could likely be optimized with parallelism. One solution for this could be to use shaders and in particular to use compute shaders[19]. Compute shaders allows for arbitrary computing of information on the GPU and as such, it is commonly used in for example particle simulation wherein individual particles can be updated in parallel.

Many papers have presented ways of generating vegetation and most of them have similar core features such as an internal skeleton representation of trees and rendering methods utilizing GPU instancing. The paper *Self-organizing tree models for image synthesis*[5] presents a biologically inspired method for procedural generation of trees and shrubs. The method simulates a self-organizing growth process in which the overall shape of trees emerges from competition for light and space.

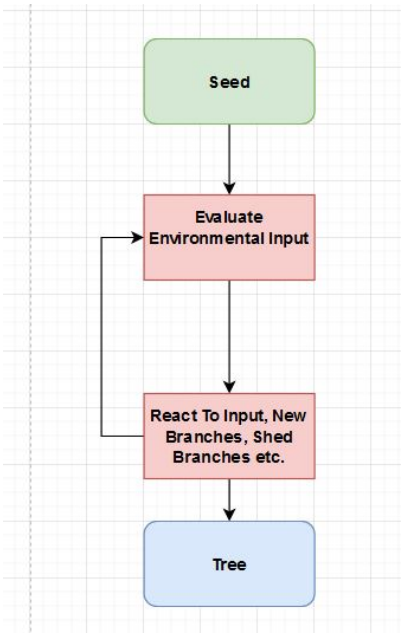


Figure 3.1: Tree Growth Engine Structure

Figure 3.1 shows the general steps of a procedural growth engine such as the one in Pałubicki et al.[5]. You start with an initial seed state. Environmental conditions are then evaluated and reacted upon by growing and adding new branches. The process is then repeated.

The internal representation of a tree can bee described as a connected graph representing the overall branching structure of the tree. A tree, T, is equal to a set of nodes, N, and a set of edges, E. Edges represents the branches and the nodes represent their connecting points. A tree can then be described as $T = \{N, E\}$. This model of a tree skeleton was adapted in this thesis. Furthermore in general three main core features from the paper was included in the algorithm implemented in this thesis. These aspects were:

- **Apical Control**
- **Shadow Propagation**
- **Pipe Model Theory**

3.1 Apical Control

The main growth loop consist of first evaluating current light condition and accumulating how much energy from sunlight that a tree gathers. The energy is then distributed through the tree following a distribution equation based on the theories of apical control. The distributed energy is then used to determine if and where any new branches should appear. An extended model of the so called *Borchert-Honda* model[20] was used. It models the regulation of growth-inducing resources which in turn controls the branching in trees. The extended model used in *Self-Organizing Tree Models for Image Synthesis* follows the formula:

$$v_m = v\frac{\lambda Q_m}{\lambda Q_m + (1-\lambda)Q_l}$$

$$v_l = v\frac{(1-\lambda)Q_l}{\lambda Q_m + (1-\lambda)Q_l}$$

for calculating the distribution of energy to main branches, v_m, and lateral branches, v_l. v_m is the distributed energy to the main branch, Q_m is the gathered energy from that main branch and Q_l is the gathered energy from the lateral branch. λ is a user defined parameter that governs how energy is distributed. A value of 0.5 will divide energy equally, where as $\lambda > 0.5$ favors distribution to main branches, and $\lambda < 0.5$ favors distribution to lateral branches. The inclusion of Q_m and Q_l essentially weight the distribution based on how much energy was gathered from the branch, and the gathered energy is a result of how much energy all of its children branches gathered.

3.2 Pipe Model Theory

In order to dynamically estimate branch diameter a version of the so the *pipe model theory*[21] was used. The theory states that branches contribute with vascular pipes through the tree and resources flow from down to up and vice versa. Thus the thickness of a branch depends on the thickness of its' children branches. Terminal branches are given a base thickness. The altered version follows the formula:

$$d^n = d_1^n + d_2^n$$

Thus:

$$d = (d_1^n + d_2^n)^{\frac{1}{n}}$$

The diameter of a branch d depends on the diameters of its main branch d_1 and lateral branch d_2. Terminal branches with no children branches are given a user defined base thickness.

3.3 Shadow Propagation

Space colonization algorithms are used by several projects mentioned in the previous works section. They are used to simulate the competition for space between branches. Every branch has an attached detection sphere. And when searching for new directions to grow this sphere can be used to ensure new branches does not collide into other branches. As an alternative to space colonization methods *self-organizing tree models for image synthesis*[5] uses a *Shadow Propagation* algorithm to find the best growth direction for new branches. The main idea is based on storing light information about the scene in order to determine where and in which direction new branches will receive the least amount of shadow. It is a faster method compared to other methods such as path-tracing that will give more accurate results. The shadow propagation method uses a voxel grid to store shadow values in the scene. Whenever a new branch is created it will first look for a direction that will yield the least amount of shadow based on the current shadow conditions. When created it will then contribute with a shadow value based on its position and it will be stored in the shadow-grid. Newly added branches contribute with shadows in a pyramidal shape downwards with decreasing shadow value further down. Though a simple and efficient method, it is likely not very scale-able since a large amount of memory will have to be stored in order for it to work for many trees in large scene.

4

Methods

This chapter gives a detailed description of the implementation process of this thesis work. The project followed three main iterations. An initial prototype created as a learning process in working with Unity, C# and HLSL as well as learning how to work with compute shaders and geometry shaders in Unity. The second prototype adapted the biologically inspired growth mechanism presented in the paper *Self-organizing tree models for image synthesis* [5] as a recursive algorithm on the CPU. With the third prototype the recursive growth algorithm was rewritten and implemented on the GPU utilizing compute shaders to handle large scale applications.

4.1 Prototype 1

Initially focus was on researching related work. Trying to understand other papers about procedural generation, rendering of vegetation in computer graphics and fast simulation techniques. Main focus with the research was find and adapt biologically inspired solutions for the growth engine. Those solutions were intended to be biologically inspired as well as compatible with the large scale intentions of the project. Such a method was then to be implemented using Unity. As an initial process it was implemented purely as a CPU algorithm. The reasoning behind firstly implementing a purely CPU-algorithm was both to be a learning process as well an easier way to test and construct a sound biologically inspired growth engine to later be implemented on the GPU. The general approach for implementing the growth engine were a somewhat iterative approach wherein a more biological accurate and efficient algorithm was developed over time. Tropisms such as gravitropism and phototropism was introduced at a later stage as well as a light evaluation method.

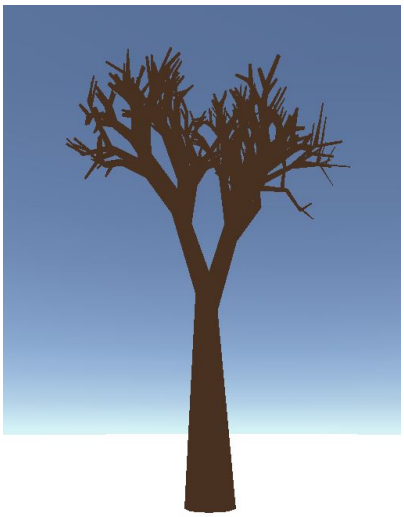


Figure 4.1: Tree from the first prototype

The initial tree generator implemented mainly focused on utilizing the basics of Unity and C# scripts by manually adding the vertices of triangles to be rendered by learning how to utilize the built in components of Unity's rendering system. All objects that are rendered in a unity scene has an attached Mesh Renderer and Mesh Filter. The Mesh Filter holds a reference to a mesh which in turn holds a list of all vertices to be rendered. In order to manually create custom meshes the Mesh Filter of an object can be accessed in a script and its mesh can be manually adjusted by giving it a custom list of vertices and triangles to render. The initial algorithm recursively created branch segments in the shape of cuboids in order to create somewhat tree like meshes.

Later on basic biological features such as gravitropism and phototropism was introduced, as well as rendering the branch segments as cylinder meshes instead. Tropisms could easily be modeled as a directional bias towards the light source in order to simulate the effect of phototropism, and a directional bias against the direction of gravity to simulate the effect of gravitropism.

After implementing this initial growth engine, focused shifted to implementing a biologically inspired growth algorithm. To achieve this several papers were studied and their respective solutions for biologically inspired growth algorithms. among them *Synthetic Silviculture: Multi-scale Modeling of Plant Ecosystems*[3] and *Self-Organizing Tree Models For Image Synthesis*[5]. The approach from the first paper was mainly adapted. The reason for choosing their approach was that it was not overly mathematically complex and their growth engine was deemed reasonably simple and flexible. Their approach was also used as a foundation in *Synthetic Silviculture: Multi-Scale Modeling of Plant Ecosystems* and it had been cited in several other studies in the field, which strengthened the belief that it would serve as a good foundation for a the algorithm. In general the goal with the thesis was not to find and implement the most realistic and biologically accurate solution for simulating plant growth, but rather find and implement one that both captured biological aspects while being simple and suitable for large scale. An overly complicated growth

engine would have taken far more effort to implement and adapt for large scale.

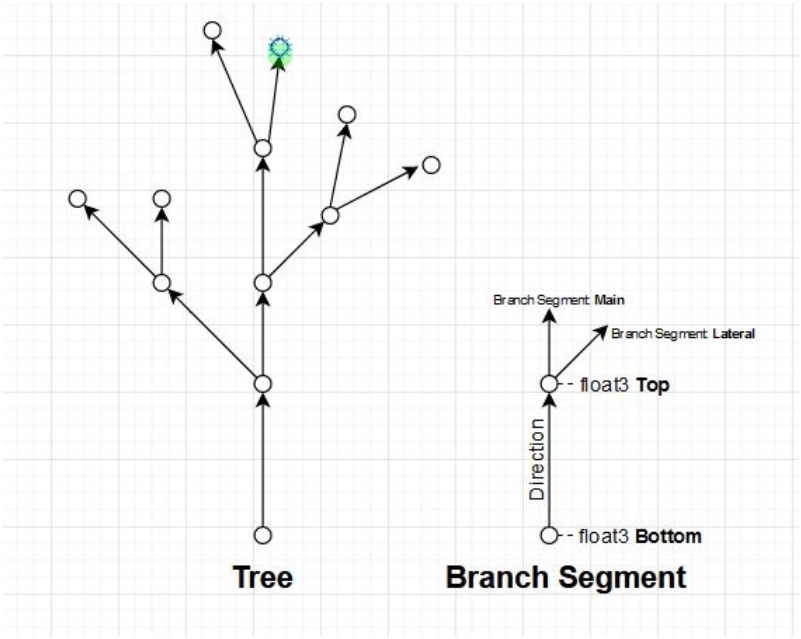


Figure 4.2: Tree skeleton graph

There needed to be an internal representation of a tree when working with it and growing it with the growth algorithm in the form of an abstraction layer between the list vertices and the branching structure of the tree. As such an approach followed by several papers from the previous work section, in which a tree could be represented as a connected graph with a set of nodes and edges was adapted. The group of nodes would consist of all branching points in the tree as well as terminal endpoints and the starting root point. These points would be connected by a set of edges which would represent the branches of the tree. From this simple representation a tree mesh could be visualized by rendering cylinders as the branches and simply modifying the rotation, scaling and position to align the cylinders to form the tree. Along side the first prototype of the growth algorithm compute shaders and geometry shaders were researched and how to utilize them using unity. They were however not used later in the project, and geometry shaders remains a tool that likely can be used for future work and improvements.

Figure 4.3: Tree with Gemoetry shader grass

As mentioned above the initial prototype manually created meshes, where triangle vertices were manually generated and added to a mesh. Initially cuboids were created, with 8 vertices. But later on a custom cylinder generator was added. The method would create one top and one bottom polygon of desired degree, and then connect the two polygons to form branch segments.

Some time was also put into adding leafs to the prototype, mainly to improve the appearance of the trees. This was done by creating a list of leaf positions consisting of every end point of terminal branches. At these leaf positions a quad was manually created and vertices were added to be rendered with a green nuance and slight transparency. However leafs remain a subject for future work.

Figure 4.4: Primitively leafed trees

4.2 Prototype 2

During the next phase of the project a more biologically correct growth engine was implemented. This included biologically inspired energy distribution for the tree growth, as well as dynamically adjusted branch diameters calculated from the pipe model theory and lastly a light evaluation scheme that would give the trees a light seeking characteristic in the direction of branches. All of this was adapted from the paper *Self-organizing tree models for image synthesis*[5].

```
public class Tree
{
    public Branch root;

    public class Branch
    {
        public Tree tree;
        public Branch main;
        public Branch lateral;

        public Vector3 bottom
        public Vector3 top
    }
}
```

The skeleton representation of a tree was implemented in a similar fashion as demonstrated by the above code snippet. An outer Tree class which acted as a container for connected branches, represented as an internal class. The internal Branch class consisted of a pointer to the next branches and its branch end points. Arguably the outer Tree class could be removed and have a tree only represented as a root Branch pointing to its child branches. However for readability and ease of understanding the external Tree class was kept.

```
public class Tree
{
    public void DistributeEnergy()
    {
        this.root.DistributedEnergy = GatheredEnergy * c;
        this.root.DistributedEnergy();
    }

    public class Branch
    {
        public void DistributeEnergy()
        {
            if (branch is terminal)
            {
                return;
            }
            else
            {
                float MainDistEnergy = g(this.DistEnergy);
                float LateralDistEnergy = f(this.DistEnergy);

                this.main.DistEnergy = MainDistEnergy;
                this.lateral.DistEnergy = LateralDistEnergy;

                this.main.DistributeEnergy();
                this.lateral.DistributeEnergy();
            }
        }
    }
}
```

Recursive function calls were heavily used in order traverse the tree by letting a branch recursively call its children branches, starting with a call on the root branch. For example as seen from the above code snippet when using the energy distribution formula recursive function calls were utilized.

Some time and effort was put into verifying that the growth algorithm worked and responded accordingly to changes in environmental light conditions. The light evaluation which was first implemented for the growth algorithm was adapted from the Shadow Propagation method. The method uses a voxel-grid to store local shadow values for the growth space. Whenever a leaf evaluates its receiving light, a lookup will occur in the shadow grid where a higher value means a stronger shadow present which in turn decreases the amount of received light. Such a shadow grid was implemented using a key-value look-up table to the shadow values. The growth space was divided into a uniform voxel grid and a three dimensional position, which every leaf would hold. Every time a new branch were created its shadow value was according to the following algorithm updated:

$$\Delta s = ab^{-q},\ a, b > 0$$

Δs is the change in shadow for a cell and a and b are user defined parameters that scales the shadow value and q corresponds to the voxel-level relative to the voxel of the newly created branch in the pyramidial shape. This method of evaluating light is a large simplification of the actual process, but it was deemed to be able to capture the essential branching patterns that emerges when branches seek out the optimal light conditions. However this method also assumes a somewhat uniform incoming light direction coming straight from above, and a more realistic and biologically correct light evaluation method stands as potential future work.

The shadowgrid was tested both by subjectively evaluating the shapes of generated trees and trying to determine if the light seeking characteristics appeared in the branching structure. It was also tested by hard coding in a shadow block. A solid block in which the shadow value would correspond to maximum shadow, and as such the trees would be forced to grow around it. In order to find the optimal growth direction for a new branch several directional samples would be taken, starting from the previous branch direction, given a maximum branching angle, directional samples would be taken and the direction in which the least amount of shadow was found would be chosen. Sampling occurred both in multiple directions but also along those directions.

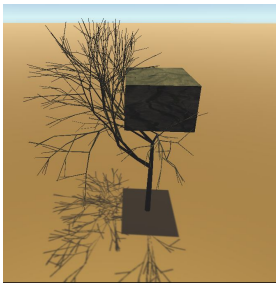


Figure 4.5

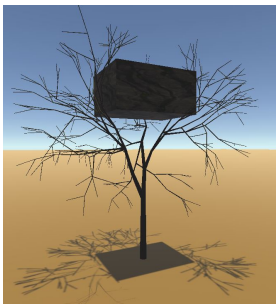


Figure 4.6

Figure 4.7

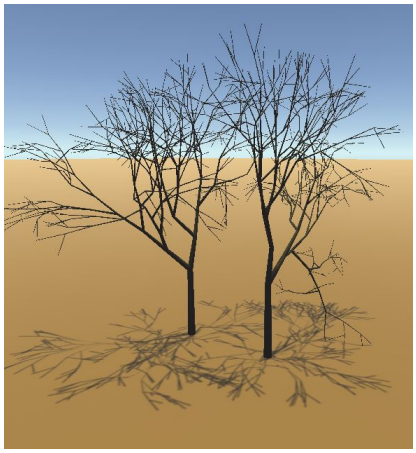


Figure 4.8: With Shadow Propagation

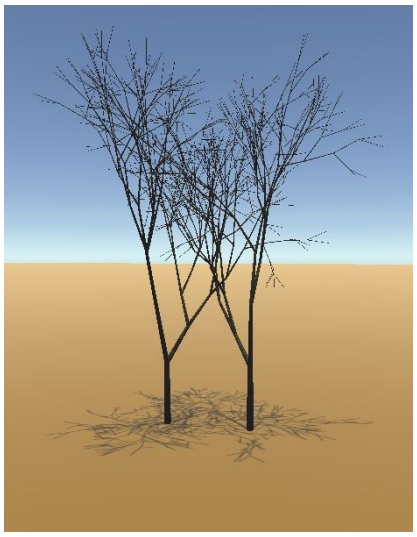


Figure 4.9: Without Shadow Propagation

Different rendering methods were also tested. The original method consisted of manually creating an entire mesh for a tree and drawing that, but GPU-instancing was later introduced since it is one of the most widely used methods for rendering many similarly shaped objects. Using every instance of that mesh's unique instance id the algorithm needed only to store one instance of the rendered mesh, in this case a cylinder, with all its vertices, as well as a unique rotation, scale and transformation matrix for every instance. because of this GPU-instancing also helped with memory storage.

Some time was also spent on trying out different parameters. Testing out different values on n to see how the branch diameters would change.

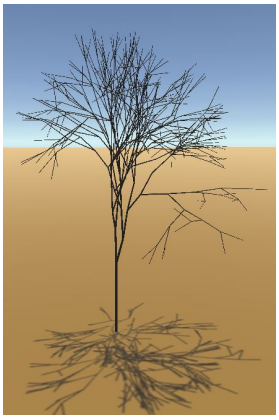


Figure 4.10: n = 3

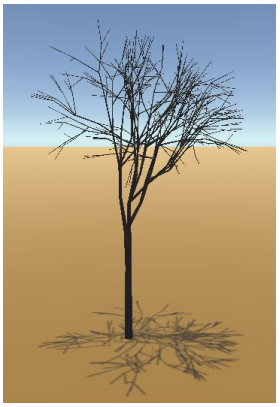


Figure 4.11: n = 2

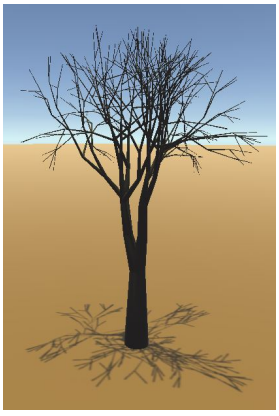


Figure 4.12: n = 3

Time was also spent on trying out different branching angles, trying out different values for the shadow propagation, different cell sizes and different sampling sizes for the branch direction sampling process. Dynamic parameters were also tested, where for example the gravitropism or the lambda value of apical control would be altered

during the growth process. It is believed that for certain trees such as weeping willows or birches the gravitropism increases over time, and as a result branches towards the end of the growth process will start pointing downwards more. And it is also believed that apical control decreases over time causing trees to spread out more and more. In nature these parameters likely differs from species to species.

Figure 4.13

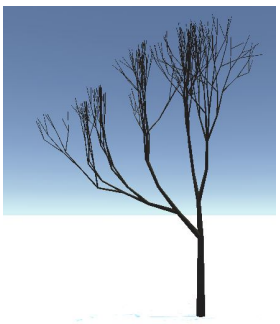


Figure 4.14

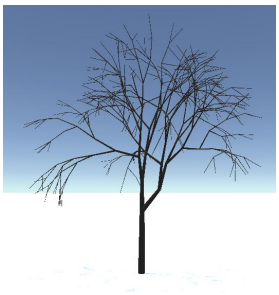


Figure 4.15

Figure 4.13 shows a tree grown with a gravitropism weight of 0.4, Figure 4.14 with a weight of 0.6 and Figure 4.15 with a declining gravitropism over time.

4.3 GPU-Version

When a viable and sound biologically inspired growth algorithm had been implemented work shifted to solving the large-scale application of the thesis problem. Evidently there are many solutions for biologically inspired procedural growth of vegetation, but adapting such a solution for large scale application would require efficient optimization techniques. As mentioned in the background the research with a focus on performance most often utilizes parallelisation. The CPU features threading possibilities to allow certain processes to be run in parallel, however not as many parallel processes as the GPU allows to be run. Thus when working with very large scale paralllizable problems and in tandem with rendering purposes it is arguably almost always favourable to utilize the GPU for such problems. For the purpose of performing arbitrarily computation not directly related to the rendering of vertices, compute shaders are a commonly used tool. For this project the intention was to speed up the growth algorithm which in turn loops over all existing trees during every iteration and grows them with respect to current environmental conditions.

Before implementing the GPU version a plan for how it could be implemented using compute shaders was created. A separate GPU version of the tree was created with a C# script responsible for controlling the overall growth process and dispatching the compute shaders. In Unity a compute shaders can be invoked using the Dispatch() method, and by specifying the size of the thread groups. Thread groups are three dimensional, with x,y,z components, which can be useful when working on three dimensional data, as the thread id associated with the thread group size is usually used to access a thread shared data buffer. The combined size of x,y,z must less than or equal to 1024. For this projects purpose only a one dimensional, x = 256, y

= 0, z = 0, thread group size was used. Data in compute shaders are often stored and passed from the CPU to the GPU using compute buffers.

One major difference when writing compute shaders is that recursive methods are not allowed in HLSL mainly due how recursive calls can quickly clog the memory. Since the CPU algorithm was implemented as a recursive algorithm, due to trees natural recursive structure, the growth algorithm had to be reworked.

```
public class Tree
{
    public Branch root;

    public void Grow()
    {
        root.GatherEnergy();
    }
    public class Branch
    {
        public Branch main;
        public Branch lateral;

        public void GatherEnergy()
        {
            if (terminal branch)
            {
                return evaluated_light_exposure
            }
            else
            {
                float Energy = 0;
                Energy += main.GatherEnergy();
                Energy += lateral.GatherEnergy();
                return Energy;
            }
        }
    }
}
```

As can be seen from the above code snippet the CPU version of algorithm gathered energy with a recursive function. From the external container class of the Tree, a call on the root branch would start a recursive chain in which all children branches would be invoked and energy would be gathered. This recursive chain had to be reworked for the GPU version. In order to replace the recursive structure of the CPU algorithm the idea was to let energy trickle down in the tree during the gathering process. By invoking the GatherEnergy compute shader every existing branch will send energy to their parent branch, and by repeating the process at least as many times as the longest chain in any tree, then all trees will have gathered energy down

to the root so that it follows the same method as used in *Self-organizing tree models for image synthesis*[5]. There is however an overhead associated with repeating the process for every branch as well as executing the process equal to the longest chain of any tree. This overhead is likely very little though since the gather energy method performs no time consuming task. The same trickle down concept could then be used for the distribution of energy, but with a reversed direction. Every branch would calculate the amount of energy to be distributed to its main and lateral branch. And by repeating the process equal to the number of branches in the longest chain it would ensure that all branches in the trees would have gathered a correct amount of energy following the energy distribution model.

The GPU-version of the algorithm was divided into five different compute shaders each responsible for a certain step in the growth simulation. The growth simulation was then executed in parallel for every branch in every tree, with each thread being responsible for exactly one branch.

- **1**: InitTrees.compute
- **2**: RecieveLight.compute
- **3**: GatherEnergy.compute
- **4**: DistributeEnergy.compute
- **5**: Grow.compute

A new **C** script was written in which all of the above compute shaders were invoked in a correct order. The InitTrees shader was executed once at the start of the simulation in order to initialize the desired amount of trees. The ReceiveLight, GatherEnergy and DistributeEnergy are all responsible for the gathering of energy and distribution of it in the trees. When that stage is finished a growth stage is executed, with the Grow shader, in which all of the lateral branches that have received enough energy will grow new branches. The way in which the trees are stored had to be reworked as well. When working with shaders large amounts of data are usually stored either in array of struct or a structure of arrays. In this algorithm due to how memory access was anticipated it was deemed best to structure the trees with several arrays. The representation of the tree was split into five arrays:

```
branch_parent_main_lateral<int>;
branch_top<float>;
branch_bottom<float>;
branch_gath_energy<float>;
branch_dist_energy<float>;
```

The *branch_parent_main_lateral* holds indexes to a branch's parent, main, and lateral branch. The indexes are stored with the parent index first, followed by the index to its main branch and lastly the index to its lateral branch. The *branch_top* array simply holds a **float3** with an x, y, z component representing the branch's top position in the growth space. The *branch_bottom* holds the position of the branch's bottom position in the growth space. The *branch_gath_energy* holds a float representing the gathered energy that a branch holds. This value is accumulated when executuing the GatherEnergy shader and it is later used by the DistributeEnergy

shader to weight how much energy is distributed to a branch's main and lateral branch. *branch_dist_energy* holds a float representing the ditributed energy which is calculated when executing the DistributeEnergy shader.

For rendering the trees a built in method called draw procedural was deemed reasonable to use. The method allows for a draw call without having to have a vertex or index buffer. A given mesh is loaded to the GPU, in this case a cylinder mesh, and then equal number of draw calls to number of branches is executed with the same mesh being rendered that many times. Every instance of the cylinder mesh is altered, transformed, rotated and scaled with a combined rotation, translation and scaling matrix. This transformation is performed in a vertex shader where every vertex uses an instance id to look up in an array of matrices which matrix to multiply each vertex with.

```
branch_trs_matrix<float4x4>;
```

The array is modified in the Grow shader, and then accessible in the vertex shader. Every time a new branch is created, that branch rotation, position and scaling is used to create the corresponding rotation, translation and scaling matrix which are then multiplied together and stored in the *branch_trs_matrix*.

For the light propagation of the GPU version of the algorithm it was decided to be different from the one implemented on the CPU, mainly due to speed and memory requirements since the shadowgrid solution would require far too much memory for the large scale purposes. The light propagation for the GPU version of the algorithm was decided to be implemented using shadowmaps. Instead of using the Shadowgrid where incoming light/shadow could be read from a grid, the shadow value would be read from a shadowmap, or several shadowmaps that would be updated dynamically. However shadowmaps are often only used for one purpose which is to cast shadows, thus the functionality for it in Unity is very tailored to that specific use case. In the most common case a shadowmap is accessed in the fragment shader or the vertex shader. In Unity they are both written in the same file, a .shader file. However a compute shader is written in a seperate file called .compute file. Thus there was initially difficulty in figuring out how a shadowmap could be generated and then accessed in a compute shader. However by manually placing a new camera in the scene, and telling it to render depth values to a texture. That texture could then be passed to the compute shaders.

A shadowmap is a 2D-texture of the scenes rendered from the perspective of a light source. The texture only contains depth values, the distance from the light source to the closest object in the scene, for every pixel in the shadowmap. This depth value can then be used when rendering the scene to cast shadows, by looking at the shadowmap and deciding if a vertex is in shadow or not, by comparing depth values with the shadowmap. The idea was to generate a couple of shadowmaps representing different times of the day where light would be coming from different directions. Given a 3D sample position it had to the transformed using matrix transformations to find the correct 2D sample position in each shadowmap. By multiplying a leaf position with the correct transformation matrix you get the correspondning screen

space positions of that coordinate in the range of [-1, 1]. However UV coordninates are in the range of [0, 1] so in order to convert it into that range you multiply it by 0.5 and add 0.5. The idea was then to let the sampled shadowvalue adjust how much light that was gathered from a specific leaf.

4.4 Evaluation Methodology

With the implemented GPU version of the tree simulation and visualization. The rendering times for the GPU version of the trees were measured with different number of branches. The results for the rendering times were gathered on a lower end laptop and more powerful desktop. Two different types of meshes with varying amounts of vertices were rendered as the branch segments in order to see how the frame rate would be affected by large amount of branches, as well as a high resolution mesh per instance. Unity has several built in meshes that were easy to try out, and the cylinder mesh and a capsule mesh were tried. The cylinder mesh had 88 vertices whereas the capsule mesh had 550 vertices.

In order to measure the simulation time of the growth algorithm, two measures were taken in order to give an idea of its performance. Firstly the simulation time was measured from the CPU in the main C# script by measuring the time before and after a frame of simulation of the growth. This measurement was complimented with a measure of how long the actual compute pass, that is the exectution time of the compute shaders. This results was gathered by using the RenderDoc[22] software.

5

Results

A biologically inspired algorithm for procedural generation of vegetation with a focus on large-scale application was implemented. The results presented were measured on the final iteration of the growth engine that was implemented on the GPU using compute shaders. The rendering times were measured as well the simulation time of the algorithm.

In terms of memory storage the implemented algorithm used the following arrays for every branch:

- *branch_parent_main_lateral*: 12 bytes
- *branch_bottom_top*: 24 bytes
- *branch_gath_energy*: 4 bytes
- *branch_dist_energy*: 4 bytes
- *branch_trs_matrix*: 64 bytes

This amounts to a total of 108 bytes per branch. As a very rough estimate a tree with approximately 1000 branches would amount to about: 0.1Mb per tree. Which in game with for example 100 000 trees would require a total storage of 10Gb.

5.1 Biological Growth Mechanisms

The biological features that were implemented in this thesis consist of the energy distribution formula, the light evaluation and light seeking characteristics of trees as well as the pipe model theory inspired branch diameters.

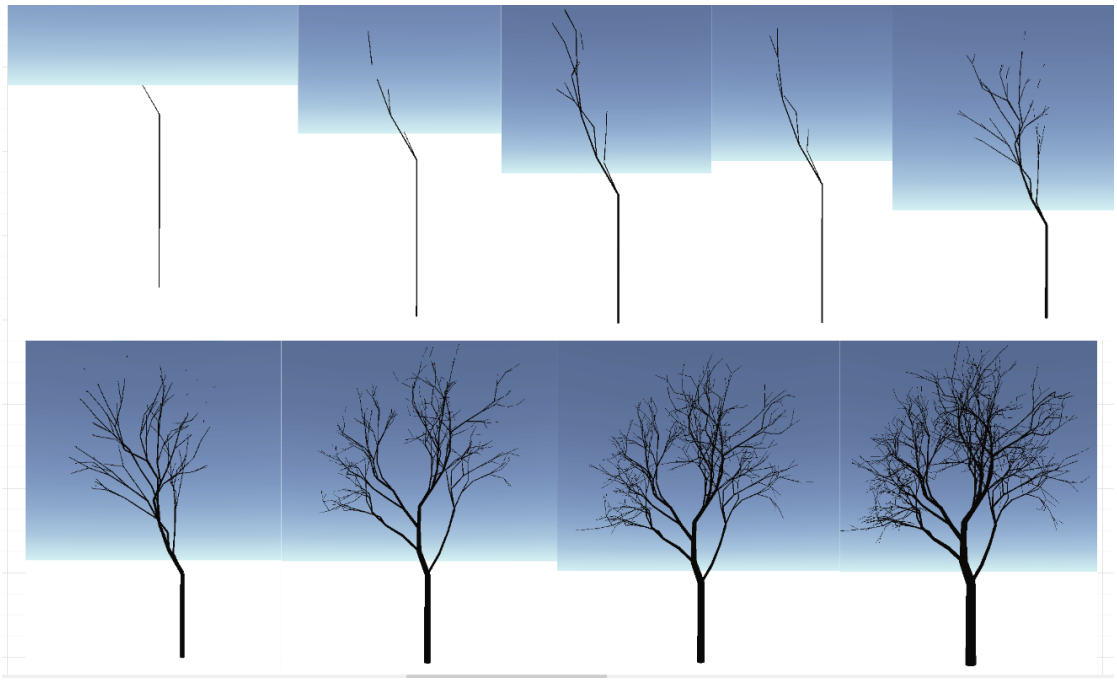


Figure 5.1: Growth Sequence

Figure 5.1 showcases the growth process of a tree with approximately 1000 branches

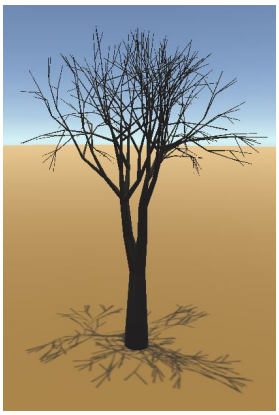


Figure 5.2: n = 1.5

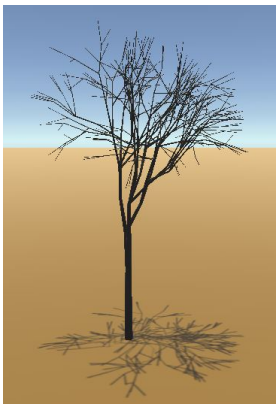


Figure 5.3: n = 2

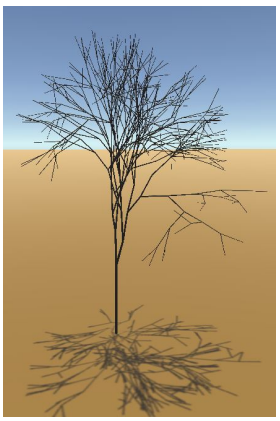


Figure 5.4: n = 3

As can be seen from **Figure 5.2, 5.3, 5.4** above, a branch's diameter depends on the number of children branches associated with the branch. The the lower value of n the faster the increase in thickness with increasing number of children branches. The value of n most likely varies between different species of trees.

As for the light seeking characteristics of the tree growth. The GPU version utilizes multiple shadow-maps to estimate the amount of incoming light.

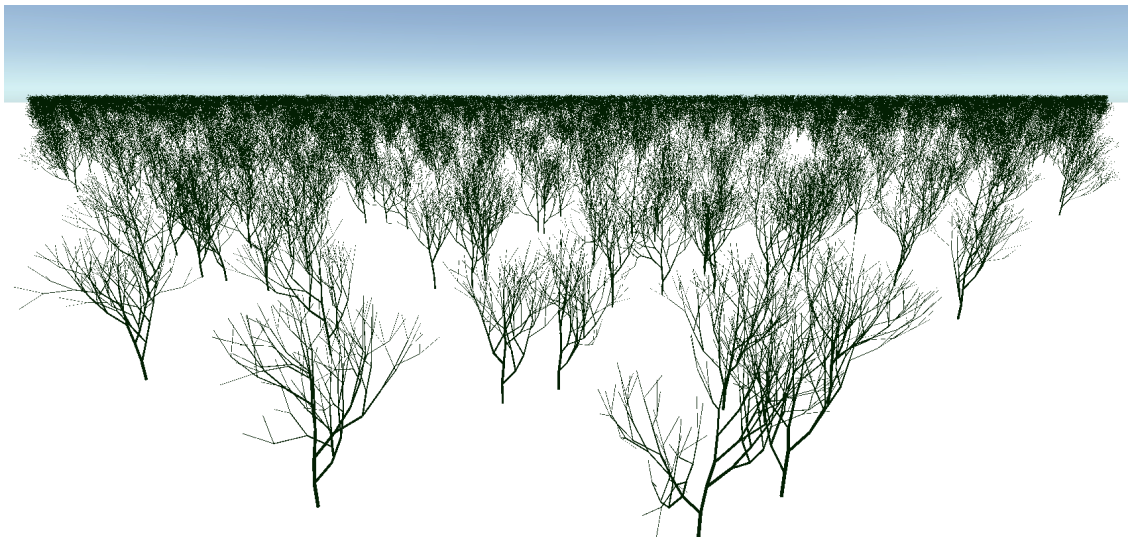


Figure 5.5: 10000 GPU trees

Figure 5.5 shows 10000 trees generated using the GPU algorithm

Lastly the energy distribution aspect implemented followed the extended borchert honda model of energy distribution. As can be seen from the images below an increasing value of λ yields a more dominant main trunk branching structure.

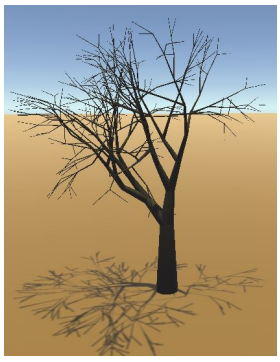


Figure 5.6: $\lambda = 0.5$

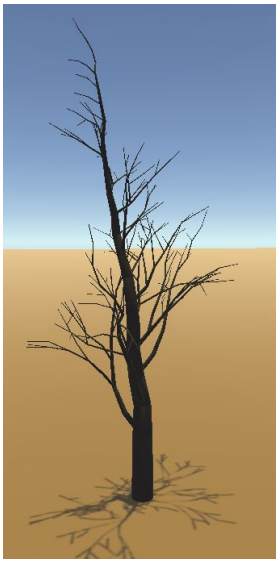


Figure 5.7: $\lambda = 0.6$

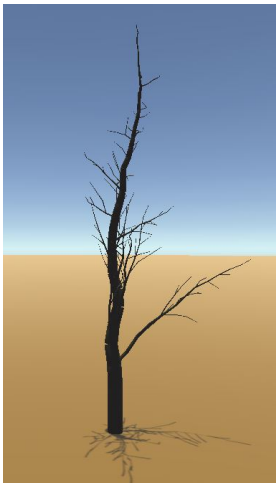
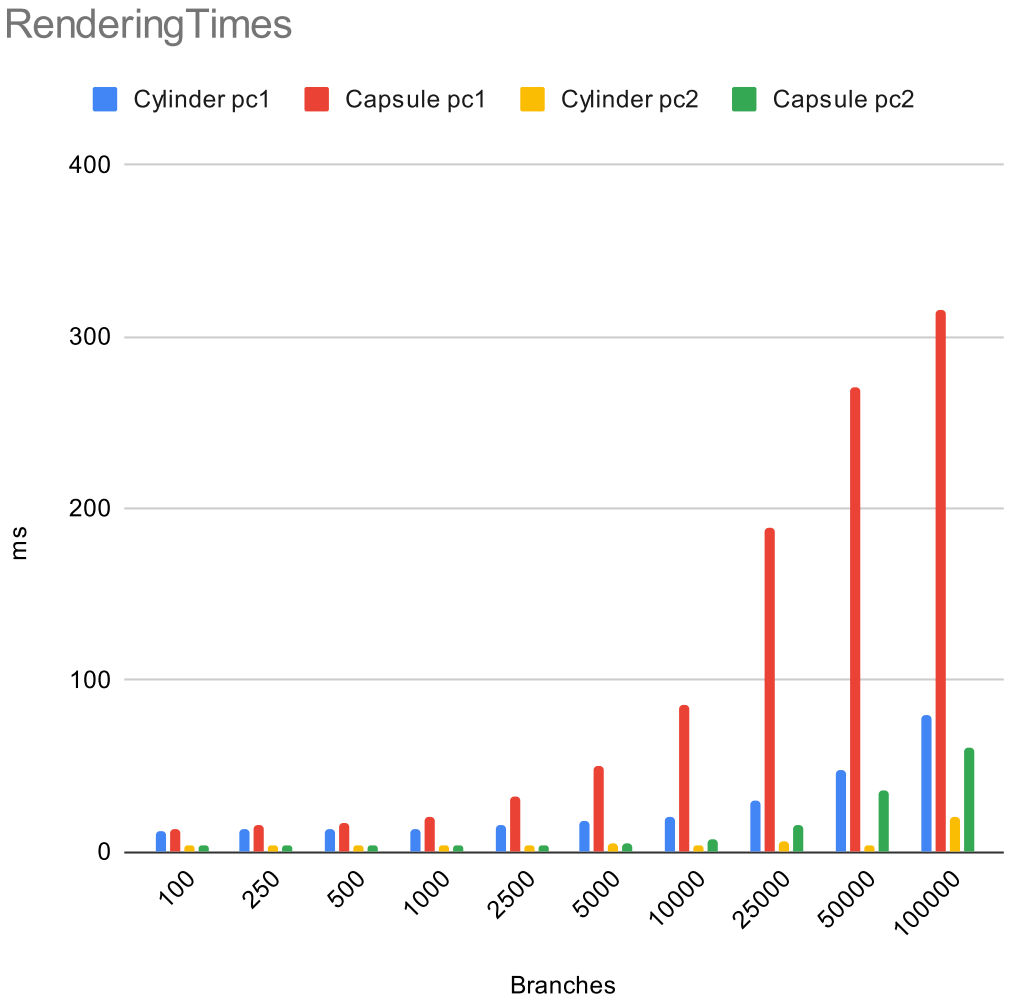


Figure 5.8: $\lambda = 0.65$

Overall it can be argued that the simulated tree growth encapsulates some properties of naturally grown trees, however many aspects and more accurate methods would be interesting and relevant to explore.

5.2 Rendering Times

Below the rendering times of the GPU growth engine are presented. The results were measured using different hardware as well as different meshes. PC1 refers to the laptop which the method was tested on and PC2 refers to the more powerful desktop. The growth algorithm was implemented in Unity, using C# and DirectX, with shaders written in HLSL. The laptop has a built in graphics card, Intel HD Graphics 5500, 128MB VRAM, with an Intel Core i5 2.30GHz and 16GB RAM. The desktop has an Nvidia Geforce 660 with 6GB VRAM and an AMD FX 8350 4.0GHz and 16GB RAM. The Table lists the measurements presented in the graph

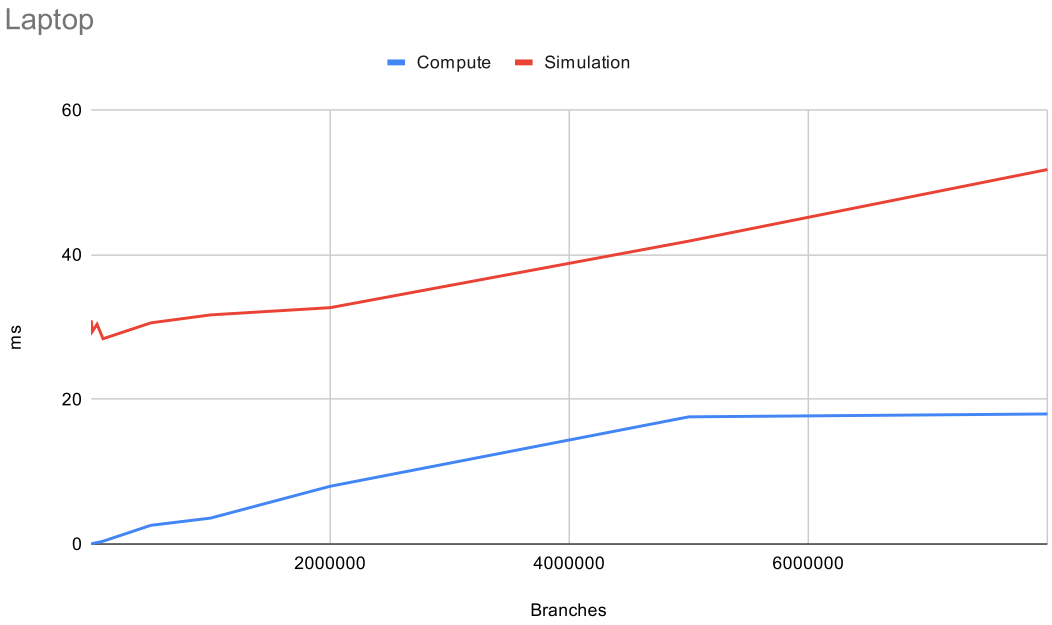


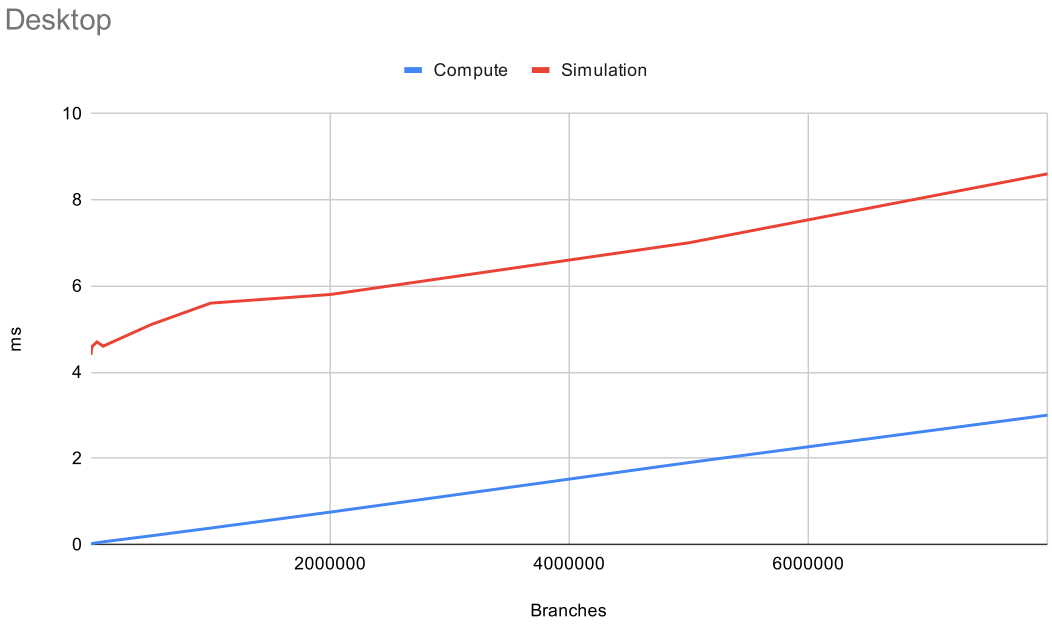
Rendering Times				
Branches	Cylinder PC1	Capsule PC1	Cylinder PC2	Capsule PC2
100	12.3ms	12.6ms	4.1ms	4.2ms
250	13.4ms	15.3ms	4.1ms	4.1ms
500	13.3ms	16.4ms	4.1ms	4.1ms
1000	13.6ms	20.1ms	4.2ms	4.1ms
2500	15.1ms	31.6ms	4ms	4ms
5000	17.4ms	49.7ms	4.4ms	4.4ms
10000	20.4ms	85.8ms	4.1ms	7.6ms
25000	30.2ms	188.6ms	5.5ms	15.8ms
50000	47ms	270ms	4.1ms	35.6ms
100000	315.4ms	79.8ms	20ms	60.3ms

Looking at the data above it show a clear correlation between rendering times in relation to number of rendered branches. A clear increase in rendering times can be seen when rendering the same amount of branches but with higher resolution mesh. There is a clear difference between the rendering times of the Laptop vs the Desktop, which is not surprising considering that the laptop has a integrated graphics which is significantly slower than the standalone dedicated graphics card of the desktop. The same correlation with rendering times and number of branches as well as the difference between the cylinder mesh and capsule mesh can be seen. As can be seen from the diagram the hardware of the desktop out preforms the laptop. The same pattern can be seen with increasing rendering times with increasing number of branches which in turn is an increase of vertices to render. However the graphics hardware of the desktop manages to render up to 50k branches with the cylinder mesh before noticing any performance impact, whereas the rendering of capsule meshes starts to impact performance at about 5k branches. Worth noting is that neither of the computers used for measuring the rendering times are up to today's standard in terms of graphical hardware, with the Desktop having a graphics card that was released in 2012. Newer hardware can be expected to yield higher rendering times.

5.3 Simulation Times

Below are the simulation times as measured on the CPU as well as execution time of the compute shader of the growth algorithm. The table lists the data as presented in the graphs.





Simulation Times				
Branches	Simulation pc1	Compute pc1	Simulation pc2	Compute pc2
100	29.5ms	0.03ms	4.4ms	0.02ms
1000	30.9ms	0.03ms	4.5ms	0.02ms
10000	29.4ms	0.07ms	4.6ms	0.02ms
50000	30.4ms	0.21ms	4.7ms	0.04ms
100000	28.4ms	0.4ms	4.6ms	0.06ms
500000	30.6ms	2.6ms	5.1ms	0.2ms
1000000	31.7ms	3.6ms	5.6ms	0.38ms
2000000	32.7ms	8ms	5.8ms	0.75ms
5000000	41.9ms	17.6ms	7ms	1.9ms
8000000	51.8ms	18ms	8.6ms	3ms

There is an obvious time difference between the actual simulation of the growth algorithm executed in compute shaders, and with the measured simulation time on the CPU. This likely is a result of other necessary operations being performed apart from the compute shader code such as sending some data from the GPU to the CPU, such as current number of branches, as well as number of branches in the longest chain. The laptop clearly performs slower than the desktop, but regardless both computers need very little time to execute the growth process of the trees. Overall a few ms for simulating tree growth in over several million branches, is arguably a very low cost for a large scale simulation not intended to be run every frame

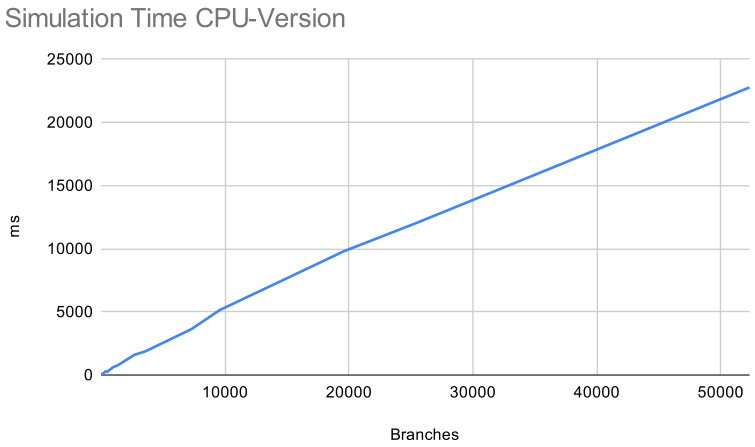


Figure 5.9: Simulation times of the cpu-version of the growth algorithm

In comparison with the cpu-version of the growth algorithm the required time only for up to 50k branches exceeds 20 seconds, highlighting the great time save that a parallel algorithm solution gains, and the how infeasible it would be to have it as non parallel sequential algorithm.

6
Discussion

6.1 Rendering Times

Main focus with this thesis was not to optimze the rendering part of the problem. GPU-instancning was used, and as can be seen from the results up to 50k branches could be rendered on the desktop within reasonable framerates if the aim is around 60fps. Exactly how many branches that needs to be rendered for a scene depends on the application. Furthermore by introducing LOD techniques more than 50k branches can arguably be rendered on screen at the same time.

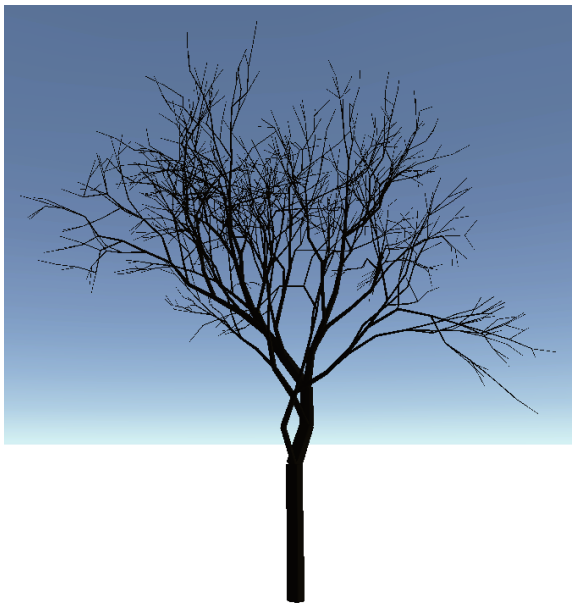


Figure 6.1: 1000 branches

Figure 6.1 shows a tree with approximately 1000 branches. Worth noting is that very small branches are often hidden in video games when leafs are also rendered, thus the addition of leafs can affect the desired amount of branches for trees.

6.2 Simulation Times

By comparing with a modern game Ghost of Tsushima[23] features over 3 million procedurally generated trees. Though not three million trees were generated and

simulated in this paper, since the algorithm was able to simulate up to 8 million branches in only a few milliseconds and there appears to a somewhat linear increase in simulation time with increasing number of branches, then simulating tree-growth for say 3 million trees each with 1000 branches would roughly amounts to a simulation time of 1 second. However that is a very rough estimate, and 1000 branches per branch is likely an over estimate of the number branches per tree. However even if the simulation time would exceed a couple of seconds, since the simulation was never intended to be run every frame, and the simulation could likely be queued and executed in parts over the course of multiple frames, thus the simulation of up 3 billion branches could likely be updated without any long in game waiting time. If the simulation is spread out and run in parts in between frames, then the simulation could likely be performed with no waiting time and no dip in FPS. However Ghost of Tsushima did not simulate any tree growth, but it gives a an estimate of about how many trees that could be expected in a modern open world game, arguably that number is very flexible though.

In order to increase the reliability of the performance measurement more measurements should take place. furthermore more accurate measurement of the growth algorithm should be done in order to isolate which parts of the algorithm that are most time consuming.

6.3 Memory

As presented in the Results. An approximation of how much a storage would take up was around 0.1Mb per tree. Which in a game with 100 000 trees would require 10Gb of storage. That could be considered a reasonable amount for a modern game, where many modern titles required up to and over 100GB of storage. However there are many optimisations that can be done in order to decrease the amount of storage per tree needed. For example the branch_gath_energy could possibly be merged with the branch_dist_energy. The branch_bottom_top could be changed to only include top. Whereas the bottom position could be fetched by accessing the parents top position. The root branch would be a special case. Last but not least the float values could potentially be changed into half values, which would decrease accuracy but every such value would take up 2 bytes instead of 4. All of these suggested optimization would bring down the total storage amount of one branch from 108 bytes to 82 bytes which would result in about 8gb of storage needed for 100k trees. There likely exist further optimizations that can be done. In any case this approach most likely uses far less space than storing unique and entire meshes of 100000 trees.

6.4 Ethical Issues

One aspect of making game development and in particular trees in games more automatised is that it reduces the need for 3D modelling artist and in turn could leave them without a job. Overall replacing manual labour with automatised solutions, could affect the job market. However making automatised solution will require the

work of programmers, and thus could lead to new job opportunities for programmers instead. Simulating tree growth and plant interactions requires thorough knowledge of real world ecosystems. Being able to simulate such systems could be useful in order to understand how they work and what can disrupt them, which in turn can help in understanding how to maintain stable ecosystems in our world.

6.5 Future Work

Though the biological aspects of the growth algorithm was implemented with a basis in biological theory. It is still a simplification of the actual and rather complex process of real plant growth. And in order to validate the results one could potentially perform tests by comparing generated trees to actual trees, or scanned models of actual trees and measure how they differ. Subjective user tests could be performed in order to determine if the growth algorithm generates visually appealing trees.

As of now the rendered branches in the trees does not overlap correctly, since only cylinders are rendered. Between every branch there will be gaps. Furthermore the branch diameters are not matched between branches. And there is a step wise decrease in thickness whereas a gradual decrease might be more correct. However these artifacts are arguably not noticeable for trees far away, but for trees that should be viewed up close it will be rather noticeable.

For the biological aspects of the growth algorithm it could be improved upon. More tropisms can be added. Other potentially more accurate light evaluation method could be explored. One aspect not touched upon much in this paper is the procedural distribution of trees. In this thesis trees are only placed using simple Possion distribution. However plant seed distribution is a research area in itself that holds interesting methods and algorithms for distributing plants in natural patterns. There are many biological aspects that could be introduced to improve the realism of the simulation. Seasonal growth, resources such as water, ground quality etc.

Another aspect not touched upon much in this thesis is the rendering of leaves. Leaves arguably play a very important role in making trees look realistic and rendering leaves is a very complex task due to their small but complex shapes and semi transparent material. Both the placements and the rendering in terms of shape and illumination effects pose as interesting and difficult problems. The paper *Real Time Rendering of Leafs*[17] could potentially serve as inspirations for leaf rendering.

Overall the implemented algorithm can most likely be optimized in a number of ways. As mentioned in the implementation section the compute shader performs the same task multiple time from the trickle down method. There might be better and faster ways of executing the gathering and distributing energy part of the algorithm. Furthemore some of the CPU to GPU readbacks could potentially be avoided and as such the simulation time of the growth would be closer to to measured executution time of the compute pass, which would be a drastical improvement.

Apart from LOD techniques many games often used a culling system where only objects in front of the player as well as in the vicinity of the player are rendered. This

is often achieved by partitioning the virtual world into chunks and only rendering one or a couple of them at a time. Such a system would likely be beneficial to be introduced for this method.

By introducing LOD techniques the rendering times could be significantly be lowered, and as a result larger quantities of trees could likely be rendered. An interesting approach that is used in Pirk et al.[3] is the use of branch modules. This approach could possibly be adapted in this method. Instead of generating trees by creating individual branches, one could instead generate trees by adding different branch modules which consist of several branches. As seen in the paper it yields rather realistic results, and performance wise this could be used to lower the number of instancing calls when drawing the trees.

7
Conclusion

The purpose of this thesis was to find solutions for a biologically inspired procedural growth algorithm intended for large scale purposes. Such solutions were explored and one was implemented on the GPU using compute shaders. The algorithm were shown to be able to simulate up to 8 million branches in a matter of milliseconds and the rendering was able to render up to 50k branches with reasonable frame-rates for interactive applications. Whether or not it can be used in a video awaits to be seen and requires further testing and measurement of the performance. As for the growth algorithm many improvements and optimizations can still be done. The biological aspects can be further explored and improved upon to heighten the realism. LOD techniques should be introduced to adapt the algorithm for use in an interactive applications with the intention to render many trees. Overall the measured performance and rendering times shows promising results that indicates that the approach presented in this thesis could be adapted and utilized in a real time interactive application such as a video game.

Bibliography

[1] xFrog, "xfrog." `https://www.xfrog.net/`. Accessed: 2022-05-31.

[2] Mojang, "Minecraft." https://www.minecraft.net/en-us. Accessed: 2022-05-31.

[3] M. Makowski, T. Hädrich, J. Scheffczyk, D. L. Michels, S. Pirk, and W. Pałubicki, "Synthetic silviculture: Multi-scale modeling of plant ecosystems," *ACM Trans. Graph.*, vol. 38, July 2019.

[4] S. Pirk, T. Niese, T. Hädrich, B. Benes, and O. Deussen, "Windy trees: Computing stress response for developmental tree models," *ACM Trans. Graph.*, vol. 33, nov 2014.

[5] W. Palubicki, K. Horel, S. Longay, A. Runions, B. Lane, R. Měch, and P. Prusinkiewicz, "Self-organizing tree models for image synthesis," *ACM Trans. Graph.*, vol. 28, jul 2009.

[6] L. Learning, "Plant growth." https://courses.lumenlearning.com/suny-wmopen-biology2/chapter/plant-growth/. Accessed: 2022-05-31.

[7] The Editors of Encyclopaedia Britannica, "Tropism." https://www.britannica.com/science/tropism. Accessed: 2022-05-31.

[8] B. F. Wilson, "Apical control of branch growth and angle in woody plants," *American Journal of Botany*, vol. 87, no. 5, pp. 601–607, 2000.

[9] "Primary and Secondary Growth in Stems." https://bio.libretexts.org/@go/page/13747. Accessed: 2022-05-31.

[10] P. Prusinkiewicz and J. Hanan, "Lindenmayer systems, fractals, and plants," in *Lecture Notes in Biomathematics*, 1989.

[11] R. Měch and P. Prusinkiewicz, "Visual models of plants interacting with their environment," in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, (New York, NY, USA), p. 397–410, Association for Computing Machinery, 1996.

[12] S. Pirk, T. Niese, O. Deussen, and B. Neubert, "Capturing and animating the morphogenesis of polygonal tree models," *ACM Trans. Graph.*, vol. 31, nov 2012.

[13] S. Pirk, O. Stava, J. Kratt, M. A. M. Said, B. Neubert, R. Měch, B. Benes, and O. Deussen, "Plastic trees: Interactive self-adapting botanical tree models," vol. 31, jul 2012.

[14] B. Beneš, N. Andrysco, and O. Št'ava, "Interactive modeling of virtual ecosystems," in *Proceedings of the Fifth Eurographics Conference on Natural Phenomena*, NPH'09, (Goslar, DEU), p. 9–16, Eurographics Association, 2009.

[15] J. P. Weber, "Fast simulation of realistic trees," *IEEE Computer Graphics and Applications*, vol. 28, no. 3, pp. 67–75, 2008.

[16] B. T. do Nascimento, F. P. Franzin, and C. T. Pozzer, "Gpu-based real-time procedural distribution of vegetation on large-scale virtual terrains," in *2018 17th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, pp. 157–15709, 2018.

[17] L. Wang, W. Wang, J. Dorsey, X. Yang, B. Guo, and H.-Y. Shum, "Real-time rendering of plant leaves," in *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, (New York, NY, USA), p. 712–719, Association for Computing Machinery, 2005.

[18] Kohek and D. Strnad, "Interactive large-scale procedural forest construction and visualization based on particle flow simulation," *Computer Graphics Forum*, vol. 37, no. 1, pp. 389–402, 2018.

[19] Khronos Group, "Compute shader." `https://www.khronos.org/opengl/wiki/Compute_Shader`. Accessed: 2022-05-31.

[20] R. Borchert and H. Honda, "Control of development in the bifurcating branch system of tabebuia rosea: A computer simulation," *Botanical Gazette*, vol. 145, no. 2, pp. 184–195, 1984.

[21] K. SHINOZAKI, K. YODA, K. HOZUMI, and T. KIRA, "A quantitative analysis of plant form;the pipe model theory,1.," *JAPANESE JOURNAL OF ECOLOGY*, vol. 14, no. 3, pp. 97–105, 1964.

[22] B. Karlsson, "Renderdoc." https://renderdoc.org/. Accessed: 2022-05-31.

[23] S. Murray, "Gaming detail: There are 3 million trees in ghost of tsushima." https://www.thegamer.com/gaming-detail-3-million-trees-in-ghost-of-tsushima/, 2020. Accessed: 2022-05-31.