ISTANBUL TECHNICAL UNIVERSITY
FACULTY OF COMPUTER AND INFORMATICS

PROCEDURAL TREE MODELING AND RENDERING

Graduation Project Final Report

Caner Coşkun
150180252

Department: Computer Engineering
Division: Computer Engineering

Advisor: Assoc. Prof. Dr. Sanem Sarıel
Co-Advisor: Dr. Selçuk Sümengen

May 2023

Statement of Authenticity

I hereby declare that in this study

1. all the content influenced from external references are cited clearly and in detail,
2. and all the remaining sections, especially the theoretical studies and implemented software that constitute the fundamental essence of this study is originated by my individual authenticity.

İstanbul, Mayıs 2023

Caner Coşkun

PROCEDURAL TREE MODELING AND RENDERING
(SUMMARY)

Procedural tree modeling and rendering is a well-researched area of Computer Graphics, mostly aiming for the visual appeal of the generated trees. This project's goal is to generate procedural trees and render the generated trees with realistic visuals at interactive frame rates; implemented techniques is a mixture of previous research and practical solutions to encountered problems.

The project can be divided into 3 parts:

1. Generation and Interactions of the Trees
2. Rendering of the Trees
3. Standalone Application

Generation and interactions of the trees can be done in different ways which will be explained in detail in Section 2. This project builds upon the approaches given in Palubicki et al.[2] and combines other research which builds upon similar principles. One of the main goals of the project is generating and rendering a small-scale forest at interactive frame rates.

Rendering of the trees requires defining, texturing, and shading the tree body(trunk of the tree) and leaves of the tree. Most of the up-to-date techniques generate tree geometry as triangle meshes and render the mesh with the traditional rendering pipeline. Currently, this project uses sphere tracing[36] to render the tree body generated from quadratic Bézier curves. Although other rendering techniques might also be applied in the future to improve performance or visual aesthetics, this project aims to introduce non-traditional techniques to tree rendering. Using sphere tracing(or other parametric rendering techniques) has the added benefit of capability of rendering more detailed visuals and on the fly parameter changes without regeneration, while taking less space to store than triangle meshes.

As one of the deliverables, a standalone application was developed for a game developer or 3D modeling artist who can create trees or forests based on intuitive parameters and see the effects of the parameters in interactive frame rates. Additionally, a user survey was conducted on the target audience to decide if the application is intuitive and useful in production.

In the end, the application developed can generate a small forest in interactive framerates based on parameters that can be edited through the user interface rendered using ImGui[37]. The bodies of trees are generated as quadratic Bézier curves rendered using sphere tracing[36]. Leaves are rendered as textured quads using the instancing technique. A forest that started with 100 trees was grown until it had ~ 200 trees, shown in Figure 1.1c, total growth process was completed in 50 iterations and took $870ms$ in total. Figure 1.1c took $9.52ms$ to render in total and rendering trees and their leaves, including shadow rendering, took $5.45ms$.

The project was developed by C++20 and OpenGL 4.6. The standalone application requires a computer that has a modern discrete GPU that supports OpenGL 4.6 on Windows 10 operating system to have a smooth user experience.

PROCEDURAL TREE MODELING AND RENDERING

(ÖZET)

Prosedürel ağaç modelleme ve görselleme bilgisayar grafikleri alanında üzerinde birçok araştırma yapılmış bir daldır. Bu araştırmaların çoğunluğu oluşturulan ağacın ve görüntünün detayı üzerine yoğunlaşmaktadır. Bu proje, ağaçların gerçek zamanda ve gerçekçi görsellerle prosedürel şekilde oluşturulması ve görsellenmesini hedefiyle geliştirilmiştir. Kullanılan metodlar önceki araştırmalara ve gerekli yerlerde pratik çözümlere dayanmaktadır.

Proje, ana hatlarıyla 3 parçaya bölünebilir:

1. Ağaçların oluşturulması ve etkileşimi
2. Ağaçların görsellenmesi
3. Uygulama

Bölüm 2'de açıklandığı gibi, ağaçların oluşturulması ve birbirleriyle etkileşimleri birden fazla yolla yapılabilmektedir. Bu proje, Palubicki vd.[2]'de sunulan modele dayanan bir yaklaşımı kullanmakta ve benzer prensiplere dayanan diğer araştırmaları birleştirmektedir. Projenin ana hedeflerinden biri, gerçek zamanda küçük ölçekli bir ormanın üretilmesi ve görselleştirilmesidir.

Ağaçların görselleştirilmesi, ağaç gövdesinin ve yapraklarının tanımlanması, doku verilmesi ve gölgelendirilmesini gerektirir. Güncel tekniklerin çoğu, ağaç geometrisini üçgenlerden oluşan modeller ile oluşturur ve modeli geleneksel görselleştirme metodları kullanarak görsele dönüştürür. Bu proje, ağaç gövdelerini karesel Bézier eğrilerinden oluşturur ve bu eğrileri görsele çevirmek için "sphere tracing"[36](küre izleme) metodunu kullanmaktadır. Gelecekte performansı veya görsel estetiği artırmak için diğer görselleştirme teknikleri de uygulanabilir olsa da, bu proje ağaç görselleştirmede geleneksel olmayan teknikleri tanıtmayı hedeflemektedir. Küre izleme(veya diğer parametrik görselleştirme teknikleri) kullanmak, üçgen modellere göre daha az yer kaplarken daha ayrıntılı görseller ve animasyonları gösterebilmek gibi avantajlar sunar.

Ayrıca bu projede, oyun geliştiricisi veya 3 boyutlu modelleme sanatçıları için bir uygulama geliştirilmiştir. Uygulama, anlaşılır parametrelerle ağaçlar veya ormanlar oluşturabilen kullanıcının gerçek zamanda parametrelerin etkilerini görebileceği bir arayüze sahiptir. Ayrıca, uygulama hedef kitlesi üzerinde bir kullanıcı çalışması yapılmış ve uygulamanın üretimde kullanışlı ve anlaşılır olup olmadığına karar verilmiştir.

Sonuç olarak, geliştirilen uygulama, "ImGui"[37] kullanılarak oluşturulan kullanıcı arayüzünde düzenlenebilen parametrelere dayanarak etkileşimli kare hızlarında küçük bir orman oluşturabilir. Ağaç gövdeleri, karesel Bézier eğrilerine çevirilir ve küre izleme[36] metodu kullanılarak gerçek zamanda görselleştirilir. Yapraklar, dolaylı görselleme (indirect rendering) tekniği kullanılarak dokulu dörtgenler olarak görselleştirilir. 100 ağaçla başlayan bir orman, $\sim$ 200 ağaca kadar büyütülmüştür ve sonuç Şekil 1.1c'te

gösterilmiştir. Toplam büyüme süreci 50 aşamada tamamlanmış ve toplamda $870ms$ sürmüştür. Şekil 1.1c için bir kare oluşturmak toplamda $9.52ms$ sürerken, ağaçların ve yapraklarının görselleştirilmesi, gölgelerinin de görselleştirilmesi dahil olmak üzere $5.45ms$ sürmüştür.

Proje C++20 ve OpenGL 4.6 kullanılarak geliştirilmiştir. Geliştirilen uygulama OpenGL 4.6 destekleyen bir modern harici ekran kartı bulunan ve Windows 10 işletim sistemi kullanan bir bilgisayar hedeflenerek geliştirilmiştir.

Contents

1 Introduction and Problem Definition

Procedural tree modeling and rendering is a vast and complex research area of computer graphics. Trees are branching structures by their nature, this property of trees results in many different problems and vastly differing solutions. Including procedural trees, procedurally generated content is used in video games, the animation industry, and many other applications because of the simplicity of creating large amounts of original content.

Some of the existing solutions to tree modeling include; fully automated generation based on parameters, generation with human input, and generation based on scans from real trees. Similar to most of the existing techniques, this project generates trees based on the assumption that trees have a recursive branching structure. This recursive structure results in difficult control of the growth and shape of the tree. Also, performance issues resulting from the recursion become more apparent as trees get more complex and the number of trees or other inputs increase. This project aims to modify the existing exponential complexity of the algorithm to implement a modeling method suitable for high-performance generation of multiple trees with visually correct aspects.

When rendering the body of the generated trees, they can be converted to triangle meshes which can be rendered using rasterization or ray tracing, or they can be rendered as parametric structures with parametric rendering methods similar to ray casting without creating triangle meshes. While conventional rendering of triangle meshes generally results in better performance, it has a limited amount visual of detail and it is not suitable for animation. Some of the mesh generation methods assume that tree branches are curves[26], and generate the geometry by converting Bézier splines to meshes. Similar to this approach, this project uses sphere tracing[36] to render each tree branch as a quadratic Bézier curve. An example tree generated with the project is rendered is shown in Figure 3.3a and a example forest generated is rendered is shown in Figure 1.1c.

(a) Render of a generated tree's body **(b)** Render of a generated tree with leaves **(c)** Render of a generated forest

Figure 1.1: Renders of generated trees

In addition to the body of the trees, leaves are an important aspect of the visual

appearance of trees. This project distributes leaves based on the number of children a branch has, this method of distribution is rather simple and is a suitable as a future research topic. Leaves are rendered as textured quads by using instancing. A render of a generated tree with leaves is shown in Figure 3.3c.

As a deliverable, an application where the user can create trees/forests by changing parameters and other inputs and render the generated trees/forests was developed. The user can change parameters for generation and rendering from a user interface created using ImGui[37], as shown in Figure 1.2.

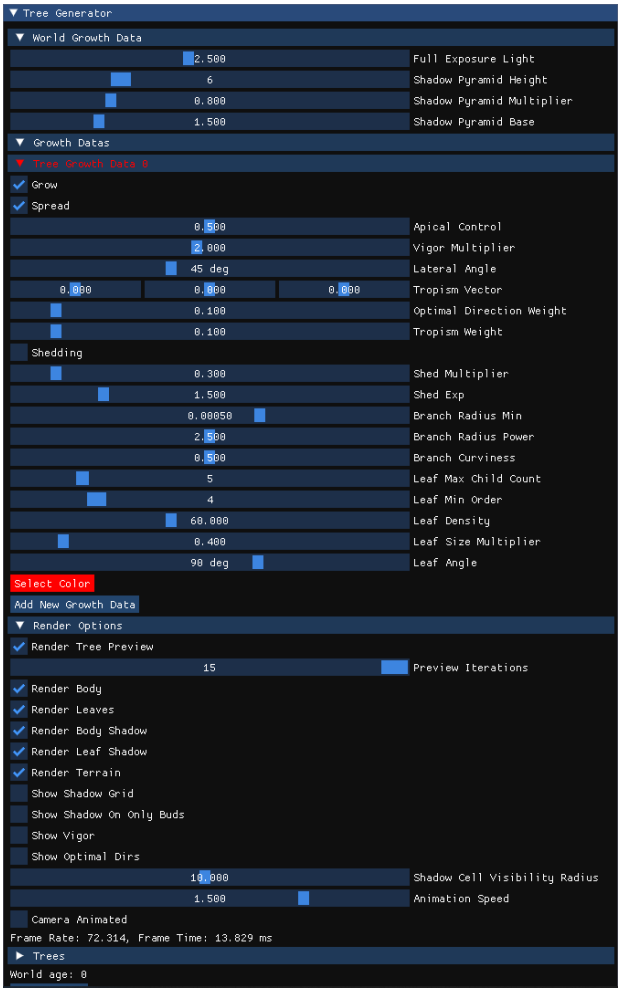


Figure 1.2: Control menu created with ImGui

1.1 Engineering Standards and Multiple Constraints

The project was meticulously structured in a modular fashion, enabling its division into distinct components that operate autonomously. Specifically, the generation algorithm and tree rendering were designed to minimize overlap, facilitating their separation. Emphasis was placed on establishing stringent ownership and accessibility relationships between classes, enhancing clarity for developers in terms of class utilization.

Considerable attention was devoted to ensuring a user-friendly experience throughout all aspects of the program. This encompassed the implementation of an intuitive user interface, streamlined tools, and optimization measures for rendering and generation processes. Furthermore, the project's development was influenced by the pursuit of incorporating novel techniques and advancements in the field, thereby impacting the decision-making process.

It is important to note that while the project prioritized high performance, the primary focus did not revolve around the inclusion of intricate visual elements such as elaborate lighting, shadows, or highly detailed trees.

2 Comparative Literature Survey

Modeling of trees has been researched for a long time, some of the early research includes papers by Honda[30]. Honda classified trees as recursively branching structures dependent on parameters such as branching angles. This classification resulted in a recursive algorithm for tree generation and created a base skeleton structure for tree generation. Later, Lindenmayer[31] created a structure for systemically modeling recursive structures named L-system. After Lindenmayer, Prusinkiewicz et al.[6] used L-systems to model trees.

On the other hand, Oppenheimer[33] argued that trees are by nature, similar to fractals and created a new model. Another method of generation uses particle systems, which was introduced to the plant generation research by Reeves and Blau[32], this method focused on the creation of an environment of multiple plants, including trees and grasses.

Ulam[34] considered trees as self-organizing structures, letting different buds of the tree have different fates, some growing to be a branch while others don't. Runions et al.[4] implemented an algorithm called "space colonization", deciding the fate of buds by the number of attraction points close to a bud. Palubicki et al.[2] created a model based on self-organization properties and used 2 different methods for calculating environmental input; a new space colonization algorithm and an algorithm called "shadow propagation", introduced by Palubicki[35]. Also, Palubicki implemented 2 different models for calculating the bud fate; Extended Borchert-Honda model[10] and Priority model. In addition, Palubicki created an interactive tool in which the user can create trees in the shape they want using the space colonization algorithm.

Takenaka[16], introduced a model to shed unnecessary buds based on the branch's radius and other factors, this model is widely used in self-organizing implementations, including this project.

This project currently uses the shadow propagation algorithm and the Extended Borchert-Honda model to generate trees. There exists alternative methods of shadow propagation, such as one that is a more performant and GPU-friendly approach introduced by Benes[20], which uses visibility from the sun through different times of the day. Although the generation of trees in this project are done on a single thread in the CPU, a similar method was implemented on the GPU by Kohek and Strnad[15] resulting in higher performance.

Generally, the rendering of trees is done by converting the generated tree structure to a triangle mesh[29]. The most basic implementation of mesh generation works by assuming the tree branches are similar to tubes and generating a triangle mesh from the assumption. Another method, presented by Nuić and Mihajlović[26] generates the mesh by treating tree branches as cubic bézier curves. Similar to this approach, this project treats branches as quadratic bézier curves.

Another method of mesh generation is based on mesh skeletonization, introduced by Hart et al.[27] and used in animation by Shek et al.[28] Although this project doesn't

create a mesh, similar to skeletonization, it renders the branches based on the distance from a point with a rendering method called sphere tracing[36]. For rendering a quadratic bézier curve using sphere tracing, the smallest distance from any 3D point to the curve must be calculable. Quilez[38] provides two different methods for distance equation to a quadratic bézier curve and calculating texture coordinates at the intersection point, which this project uses with small modifications to the calculations.

Another method of rendering bézier curves is ray tracing, which requires a ray-curve intersection algorithm. One such intersection algorithm is introduced by Reshetov and Luebke[21], which is focused on ray-hair intersections.

Rendering forests are not performance friendly because of the complex geometry of trees, to solve this problem Weber and Penn[29] focused on level of detail(LOD) methods. Since this project uses sphere tracing, a custom level of detail method was implemented that lowers rendering detail of the sphere tracing based on the pixel's distance.

3 The Developed Approach and System Model

As explained in Section 2, algorithms implemented in the project can be divided into two sections; generation of the trees and rendering of the trees. An example forest generated and rendered with the project is shown in Figure 1.1c. Passing data between the Application, generation algorithm and rendering algorithm is explained in 3.4.

3.1 Generation of The Trees

The algorithm for tree generation is based on methods described in Palubicki et al.[2] The implementation can be divided into 6 parts: calculating environment input, accumulating light received by buds, distribution of vigor to buds, adding new shoots, shed branches, updating branch radius.

3.1.1 Data Model

There are 3 main structs used by the generation algorithm:

1. TreeNode Struct

```
struct TreeNode {
    enum NodeStatus {
        BUD, ALIVE, DEAD,
    };
    vec3 startPos;
    vec3 direction;
    float length = 1.0f;
    uint32 childCount = 0;
    NodeStatus nodeStatus = NodeStatus::BUD;
    TreeNodeId id = 0;
    uint32 order = 0;
    uint32 createdAt = 0;
    //iteration parameters
    float light = 0.0f;
    float vigor = 0.0f;
    //root if null
    TreeNode* parent;
    TreeNode* mainChild = nullptr;
    TreeNode* lateralChild = nullptr;
 };
```

Listing 3.1: TreeNode Struct

A TreeNode is owned by the Tree that its part of. Each node has an *id* that is unique within its tree, a *startPos* that is also the end position of its parent, a *direction* and *length* if its status is $ALIVE$, a *parent* node if it is not a root node, a *mainChild* and a *lateralChild* if its status is $ALIVE$. *order* is the level of the node in the tree data structure(distance from the root node). *createdAt* is the iteration that the node was created at. *light* and *vigor* are parameters used while a generation iteration is ongoing, which is explained in 3.1.2. A child node(non-root) is the dominant child of its parent if the node has more children(count of nodes below the node in the tree data structure) than the other child of its parent.

2. Tree Struct

```
struct Tree {
    TreeWorld* world;
    GrowthDataId growthDataId;
    uint32 age;
    uint32 id;
    uint32 seed;
    uint32 lastNodeId;
    TreeNode* root;

    //For renderer cache
    std::vector<rb<Branch>> branchs;
  std::unordered_map<TreeNodeId, Branch> cachedBranchs;
};
```

Listing 3.2: Tree Struct

A tree store and own all of its nodes(creates and deletes its nodes). Tree's *id* and *seed* is determined by the $TreeWorld$ it was created from, its age increases by 1 in every iteration and $lastNodeId$ stores the last created node's id. $lastNodeId$ is used when a new $TreeNode$ is created, it is increased by one and the created node's id is set to the new $lastNodeId$. *root* is the root tree node which are stored as a tree data structure. $growthDataId$ is an uint32 and it is the id of growth data parameters stored in the world and used as a reference rather than storing it directly in the struct, the real growth data parameters are accessed through the growthData map in world.

3. TreeWorld Class

$TreeWorld$ contains the general information about the world, it owns and stores all the trees in the world. It contains the information related to shadowing part of the algorithm and the actual voxel shadow grid that is used in the generation algorithm. Also, it contains the growth data id to growth data map.

```
class TreeWorld {
    uint32 age;
    uint32 seed;
    ivec3 worldSize;
    vec3 leftBottomCorner;
    float cellSize;
    std::vector<float> pyramidShadowLUT;
    std::vector<std::unique_ptr<Tree>> trees;
    uint32 treeCount = 0;
    std::vector<ShadowCell> shadowGrid;
    std::shared_ptr<EditableMap> presetMap;
    std::map<GrowthDataId, TreeGrowthData> presets;
};
```

Listing 3.3: TreeWorld Class

3.1.2 Structural Model

1. Calculating Environment Input

   The environment input is calculated using the "shadow propagation" algorithm described in Palubicki[35]. The algorithm works on a 3D grid, each cell contains its shadow value as a float. Each terminal bud of the tree casts a pyramid shadow, as the height difference from the bud increases, its effect on the shadow decreases. Since each bud's effect on the world is independent, multiple trees can effect each other, as seen in Figure 3.1. It can be observed that the trees interact with each other and grow in opposite directions to each other as expected. Since shadow of a bud is calculated on a grid cell using the *pow* function, it has a considerable effect on the performance, as an optimization, a look up table for effects of a bud at a given step is precalculated. Since there is only a small amount of combinations given the shadowing parameters.

2. Accumulating Light

   Light received by a bud is calculated from the shadow value of the cell at the bud's position. Lesser shadow value at the bud's position means more light received by the bud. Each node accumulates and stores the light received by its children.

3. Distribution of Vigor

   The distribution of vigor is based on the Extended Borchert-Honda model. The BH model works by distributing the light received by the parent node to its children based on a parameter called "apical control". Lesser apical control results in more vigor distributed to the lateral branch, and higher apical control results in more vigor distributed to the main branch. If the apical control is lower than 0.5, the tree grows more like a bush. If the apical control is higher than 0.5, the tree grows more in the vertical direction. "Vigor multiplier"(α) is a user-defined

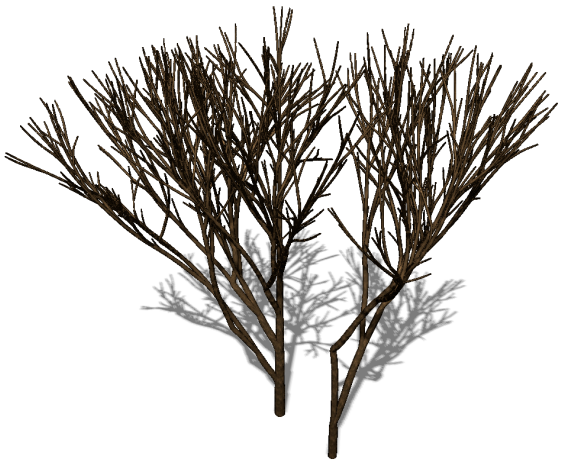


Figure 3.1: 2 interacting trees

parameter that is used to calculate the total light received at the root.

$$v_{root} = l_{root} * \alpha$$

$$v_m = v\frac{\lambda l_m}{\lambda l_m + \lambda l_l}$$

$$v_l = v\frac{\lambda l_l}{\lambda l_m + \lambda l_l}$$

4. Adding New Shoots

   If the vigor of a bud is higher than 1, new nodes will be created from the bud. The number of new nodes is calculated as $\lfloor vigor \rfloor$ and the length of the new nodes is calculated as $\frac{vigor}{\lfloor vigor \rfloor}$. The direction of the new nodes is calculated based on the weighted sum of the original direction of the bud, a tropism vector, and the optimal direction calculated from the environmental input. The optimal direction is calculated as negative of the gradient of the shadow values at neighboring cells.

5. Shed Branches (Optional)

   Each node is looped recursively starting from the root. If the value of p is negative, the node and its children are removed from the tree.

$$p = node.vigor - shedMultiplier * node.childCount^{shedExponent}$$

6. Updating Branch Radius

   The branch radius is calculated from the child count of the node. $radiusN$ is a user defined parameter, generally between 2.0 and 3.0.

$$radius = node.childCount^{radiusN}$$

7. Spreading Seeds (Optional)

   When a tree is above a user chosen age, the tree starts spreading seeds around the world. Amount of seeds that will be spread is calculated as $seedCount = random(log(root.vigor))$ and the max distance that a seed will spread is calculated as $maxDist = log(root.vigor) * spreadDistMultiplier$. A small population of trees emerging from a single tree is shown in Figure 3.2.

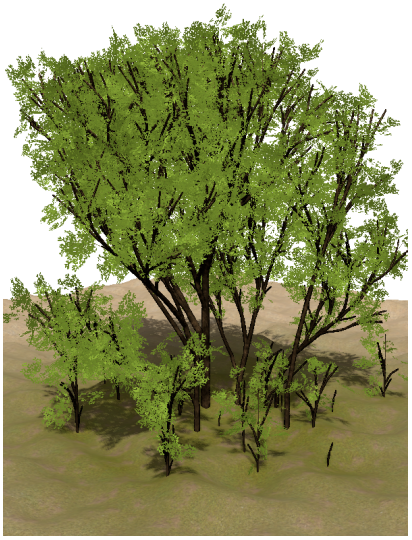


Figure 3.2: Spreading trees

(a) Apical Control = 0.46 **(b)** Apical Control = 0.5 **(c)** Apical Control = 0.56

Figure 3.3: Trees generated with different apical control

3.2 Rendering of The Trees

Rendering of the trees is divided into two, static rendering and animated rendering, animated rendering shows the growth animation of the trees.

This project can render a tree that is given as a collection of connected branches, which is generally the result of a generated tree.

3.2.1 Data Model

Animated rendered branchs and static rendered branchs have 2 different structs:

1. Static Branch Struct

```
struct alignas(16) BranchShaderData
{
    mat4 model;
    //bezier positions, actual data is vec3
    vec4 A;
    vec4 B;
    vec4 C;
    //color for debugging, actual data is vec3
    vec4 color;

    float lowRadius;
    float highRadius;

    //used for uv y offsets
    float startLength;
    float length;

    //uv x offset
    float offset;
    int order;
};
```

Listing 3.4: Static Branch Struct

2. Animated Branch Struct

```
struct alignas(16) AnimatedBranchShaderData
{
    mat4 model;
    //bezier positions, actual data is vec3
    vec4 A;
```

```
    vec4 B;
    vec4 C;
    //color for debugging, actual data is vec3
    vec4 color;
    //lowRadius is animated from animationBounds.x to 1
    vec2 lowRadiusBounds;
    //if the branch is new, high radius stays 0 until
   animationT>animationBounds, then it is animated
   between animationBounds.y < animationT < 1
    vec2 highRadiusBounds;
    //bezier T bounds, both is 1 if the branch already
   exists, animated between animationBounds
    vec2 TBounds;
    //time where animation is active, 0-0 if branch
   already exists
    vec2 animationBounds;
    float startLength;
    float length;
    float offset;
    int order;
};
```

Listing 3.5: Animated Branch Struct

3.2.2 Structural Model

As explained before, the tree body is rendered using sphere tracing to render quadratic bézier curves. The rendering algorithm can be divided into 3 parts: creating quadratic bézier curves from branches, sending data to GPU, and rendering quadratic bézier curves.

1. Creating Curves from Branches

   A quadratic bézier curve is created from 3 points, additionally, a starting radius and ending radius are needed to create a surface around the curve. Although these variables are enough to define the surface of the curve, to render the curve with textures, texture coordinates at a point on the curve must be calculated. To solve the discontinuities of texture coordinates on the connection of the branches, 3 additional variables are required: $uvOffset$, $startLength$, and $branchLength$. Additionally, the bounding box of the curve is required, as it will be explained in curve rendering part, which is stored as a 4 by 4 matrix. There are two possible scenarios for calculating the positions of the curve: the branch is the main child of its parent or the lateral child of its parent.

   (a) If the branch is the main child, then its curve's starting point is the starting point of the branch and its ending point is the end point of the branch. The

middle point of the curve is calculated as:

$$B = (parent.endPos + parent.direction * \frac{node.length}{2} + mainChild.startPos - mainChild.direction * \frac{node.length}{2}) * 0.5 \quad (1)$$

(b) If the branch is the lateral child of its parent, its curve's starting point is the starting point of its parent, the middle point is the starting point of the branch and its ending point is the end point of the branch.

Starting radius of the curve is the radius of the branch and its ending radius is the radius of its main child. $uvOffset$ is calculated as the sum of the angle between the coordinate axis' of the previous curve and the current curve and the last curve's offset. $startLength$ is calculated as sum of last curve's $startLength$ and last curve's $branchLength$. $branchLength$ is calculated as the length of the curve. The bounding box of the curve is calculated as given in Quilez[39].

If the rendering is animated, curve's parameter, low and high radius of the branch is animated. Since multiple branchs can be created from the same bud in the same iteration if vigor of the bud is bigger than 2, $animationBounds$ describes when the animation for the branch starts and ends, if branch already exists, $animationBounds = (0.0, 0.0)$.

Also, to avoid unnecessary computation, previously calculated branches and their leaves are cached in the tree and reused when the tree grows.

2. Sending Data to GPU

   Shader Storage Buffer Object[40]'s(SSBO) are used to transfer the data between the CPU and GPU. Given structs in Section 3.2.1 are used to upload curve's data from CPU to SSBO's standard layout format.

   After the SSBO is created, all of the curves of each tree are rendered as cubes using instancing.

3. Rendering Curves

   In the vertex shader, each vertex of the cube is multiplied by the curve's bounding box matrix. In the fragment shader, the intersection of the curve and the ray is calculated as given in Quilez[38]. To solve the issues related to texturing, the unclamped spline parameter is returned from the $sdBezier$ function, for distance and radius calculations, the spline parameter is clamped. The vertical texture coordinate of the is directly calculated as $branch.startLength + unclampedT * branch.branchLength$. Horizontal texture coordinate is calculated by converting the intersection point from Cartesian coordinate system to cylindrical coordinate system and summed up with the $uvOffset$.

   Surface normals of the curve are calculated as given in Quilez[38], but this results in artifacts on the connection of two curves. To solve this issue, the end and the start of the curves are assumed to be cones and the normal of the cone is blended with the calculated normal of the curve.

   In addition to the normal smoothing, normal mapping[41] was implemented to add more detail to the rendering of the branches. Bi-tangent and tangent vectors

required for normal mapping is calculated using the derivative of the curve at the closest point to the intersection point and the vector between the intersection and the curve's closest point. A close-up image of a tree branch is shown in Figure 3.4 to show the effect of normal mapping.

Figure 3.4: Close-up image of a branch

If the rendering is animated, animated parameters are calculated as given in Listing 3.6.

```
float MapBoundsT(vec2 TBounds, vec2 animationTBounds,
   float animationT) {
    return mix(TBounds.x, TBounds.y, clamp((animationT -
    animationTBounds.x) / max(0.001, animationTBounds.y
   - animationTBounds.x), 0.0, 1.0));
}

float MapLowRadius(vec2 lowRadiusBounds, vec2
   animationTBounds, float animationT) {
    return mix(lowRadiusBounds.x, lowRadiusBounds.y,
   clamp((animationT - animationTBounds.x) / (1.0 -
   animationTBounds.x), 0.0, 1.0));
}

float MapHighRadius(vec2 highRadiusBounds, vec2
   animationTBounds, float animationT) {
    return mix(highRadiusBounds.x, highRadiusBounds.y,
   clamp((animationT - animationTBounds.y) / max(0.001,
   1.0 - animationTBounds.y), 0.0, 1.0));
}
```

Listing 3.6: Animated Branch Parameter Calculation

4. Rendering the Shadows Shadows are rendered using a simple shadow mapping algorithm. Shadow mapping works by rendering the scene as a depth map from the view of the light(a directional light for this project), then the rendered depth map is used to compare the distance of the pixel's world position to the rendered depth. If rendered depth is bigger than the pixel's position's depth, the pixel is not in a shadow, if not, the pixel is shadowed by some object. Shadows of leaves are rendered as normal meshes to the depth map. For rendering the branches, they are rendered as curves as explained before but instead of creating a ray from the camera to the pixel of the bounding box, the ray is created from the bounding box's farther face to the camera to avoid rendering the near face of the curves in the shadow depth map. Since the depth map is rendered before the actual rendering starts, self shadowing is solved. An example tree rendered with shadows is shown in Figure 4.1b.

5. Rendering the Terrain

   The terrain, shown in Figure 3.5, is a mesh generated from a given height-map, which is a black and white image that each pixel represents a height. If a point on the terrain is close to a tree, its color is blended with a grass texture.

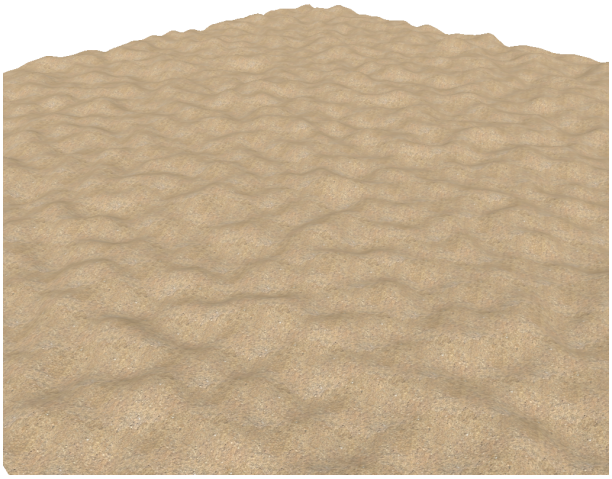


Figure 3.5: Rendered image of a generated terrain

3.3 The User Application

The user can change the generation parameters and rendering options from the application. It contains:

- A map editor which is used to change the initial distribution of the trees growth data, a distribution map that resulted with the forest in Figure 1.1c is shown in Figure 3.6, where each color represents a different growth preset.

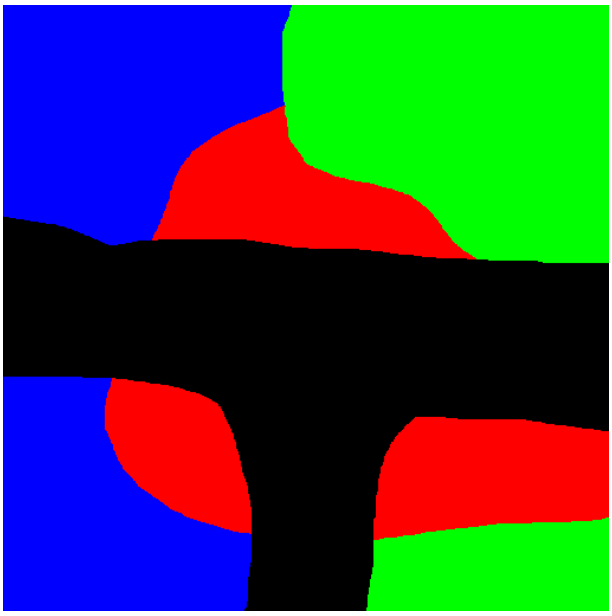


Figure 3.6: Initial distribution for forest in Figure 1.1c

- A growth preview option that iterates the world for the given amount but keeps the original world. When a parameter is changed, the previewed world is regenerated from the original world's state. Also, iterating the original world once iterates the preview world.
- A procedural terrain generator and renderer which creates a terrain from a given height-map.
- Shortcuts for iterating the world, switching to the map editor, changing directional light's direction.
- A window, shown in Figure 1.2, which the user can see and change all growth datas of the world; all of the trees' age, branch count and leaf count; rendering options and debugging options.

3.4 Dynamic Model

The UI is immediate mode, meaning that it always refreshes and almost no data is cached, only the renderers for trees are regenerated when world changes between preview and the original world.

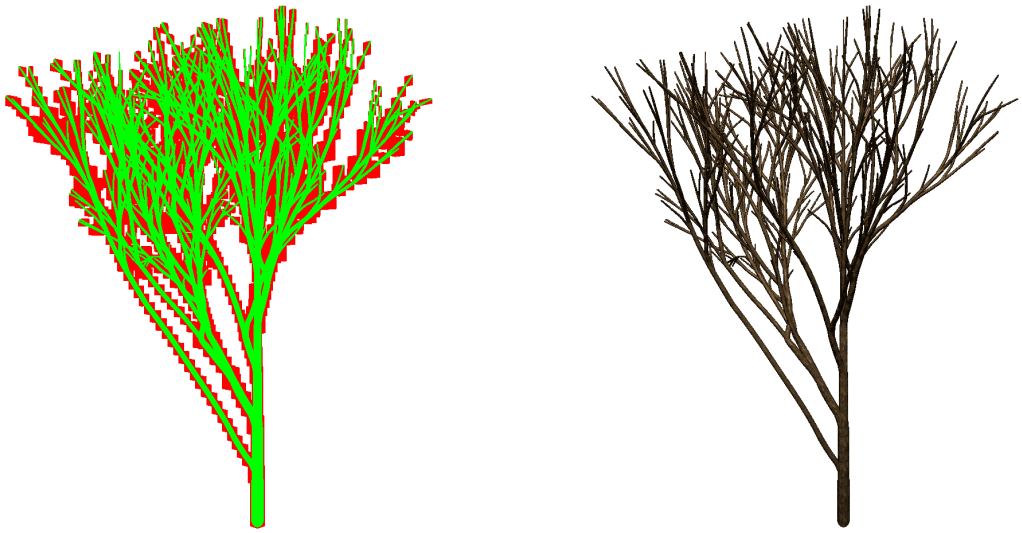
Generation takes only the growth datas from the UI and has no other input from the UI or the renderer. Generation signals UI and renderer with events. Each tree has three events; $OnBeforeGrow$, triggered before a growth iteration begins; $OnAfterGrow$, triggered after a growth iteration ends; $OnDestroy$, triggered before a tree is destroyed. $TreeWorld$ has three events; $OnAfterWorldGrow$, triggered after a growth iteration on the world ends; $OnTreeCreated$, triggered after a new tree is created, $OnTreeDestroyed$, triggered before a tree is destroyed.

Each $TreeRenderer$ subscribes to respective tree's $OnAfterGrow$ and $OnDestroy$ events. When a $OnAfterGrow$ event is triggered, it updates the buffers in the GPU with the new branches' and leaves' data and clears the buffers when $OnDestroy$ event is triggered. Additionally, animated renderers update their last updated time, which is used for the animation.

4 Experiment Setup and Design

The project was developed using C++20 and OpenGL 4.6 on a computer with Windows 10 as its operating system and a Nvidia RTX 2060 GPU. ImGui[37] was used to render the UI of the user application.

4.1 Performance Metrics



(a) Ray hit and miss image of a tree **(b)** Render of only the branches of a tree

Figure 4.1: Tree used for benchmarks

Figure 4.2: Forest used for benchmarks

4.1.1 Performance Metrics for Generation

Figure	Branchs	Total	Light	Vigor	New Shoots	Shedding
Single Tree(3.3c)	$2,410$	$5.98ms$	$1.06ms$	$0.05ms$	$3.95ms$	$0.86ms$
Two Trees(3.1)	$3,497$	$8.46ms$	$1.56ms$	$0.07ms$	$5.37ms$	$1.33ms$
Forest(4.2)	$237,551$	$1402ms$	$349.26ms$	$14.88ms$	$722.8ms$	$377.2ms$

Table 4.1: Benchmark results of tree generation

A single tree, two trees and a forest is benchmarked and the results are shown in Table 4.1. As it can be seen from the table, appending new shoots are the main bottleneck for the algorithm, mainly because of the shadow propagation. Whenever a new bud is added to the world, it needs to create shadows as it was explained before, this step needs to access multiple cells from the shadow grid and calculate its shadow value for that cell, resulting in a performance impact. To improve this steps performance, an optimization was implemented so that calculation of shadow values are faster using look up tables.

Also, it can be seen that the complexity of generation is linear and directly related to branch count. As the results show, generation times are within interactive given that they are not generated every frame, this is achieved with linear complexity of the algorithm and said optimizations,

4.1.2 Performance Metrics for Rendering

			Shaded		Shadows	
Figure	Branchs	Total	Branches	Leaves	Branches	Leaves
Only Branches(4.1b)	2,410	$0.35ms$	$0.35ms$	-	-	-
Single Tree(3.3c)	2,410	$0.50ms$	$0.29ms$	$0.11ms$	$0.06ms$	$0.03ms$
Forest(4.2)	237,551	$12.0ms$	$4.09ms$	$3.90ms$	$2.98ms$	$1.02ms$

Table 4.2: Benchmark results of tree rendering

A single frame for rendering a single tree with and without any shadows/leaves and a forest is benchmarked and the results are shown in Table 4.2. It can be seen that shadows take lesser time than shaded rendering, even though shadow map has a resolution of $2048x2048$ and the rendered images have a resolution of $1024x1024$, this is mainly because effect of LOD is increased for shadow calculations and the distance of rendered objects from light's position compared to the actual camera. Also, it can be seen from the results that as the number of branches increase, rendering times increase linearly. Additionally, if a branch is farther away from the camera, it takes less screen space, resulting in faster render times, this effect can be observed with the results of Forest(4.2) benchmarks. Also, a simple benchmark of misses resulting from ray marching are shown in Figure 4.1a, red pixels are misses which are within the bounding box but outside the branch curves, green pixels are branch curves. In conclusion, the results show that the proposed rendering method results in interactive frame-rates for tree and forest rendering.

5 Experimental Results and Discussion

- Although there exists research focused on the performance of the generation of trees, most of the research focuses on creating an end result with high detail instead of focusing on high performance. This project implements high-performance algorithms for the generation of trees and forests. The interactive generation of a single tree and small-scale forests was targeted in the interim report and was achieved as shown in Section 4.1.1.

- Most of the previous methods for generating visuals for trees or forests focus on only creating a triangulated mesh, this project uses non-conventional methods, such as sphere tracing or ray tracing to render the generated trees in high detail and in high performance. Rendering a small forest in realtime performance was targeted in the interim report and achieved as shown in Section 4.1.2. Proposed rendering method can be used alongside the existing rendering methods.

- Since the project uses parametric rendering, growth animations or similar other modifications to the tree can be rendered in realtime without pre-calculated data. Similarly, results of the procedural generation are rendered with minimal calculations, which would have required mesh generation for each iteration if the trees were rendered as triangle meshes.

- A new method was introduced to fix shading artifacts at the intersection of two connected quadratic bézier curves. Calculated surface normals on the intersection of two curves are blended to have a smooth transition between normals and resulted in smooth shading.

- A user survey was prepared and its still on going. The survey included questions about the satisfaction of results of generation and rendering. Also, it includes a quiz section that gives the user some parameters and asks the user to choose the generated tree from the given parameters. Polishing on different parts of the project will be done based on the results of the user survey.

6 Conclusion and Future Work

In the end, this project implements scalable and flexible generation and rendering methods for trees and forests that is focused on high performance. The developed user application is suitable for further development so that it can become a production ready tree generation tool. Used rendering method is a novel way of rendering trees that is suitable for animation and procedural generation while also being able to achieve high performance, used rendering method can be used alongside traditional rendering methods. To further polish all aspects of the project, a user survey is conducted.

Implemented generation model is based on the model proposed in alubicki et al.[2], in addition to the base model, this project implements tree spreading methods, user controlled forest distribution, and integration with real-time rendering. The developed model is suitable for further research, such as generation using multi-threading or GPU, using different shadow propagation methods or completely different resource calculation methods, different spreading algorithms, different leaf distribution methods. Although tree spreading algorithms were not a high priority topic for this project and the current implementation is a valid way for use in forest generation, more detailed and physically based spreading algorithms can be researched, such algorithms can be based on the properties of terrain, including nutrient properties of the dirt; properties of the weather and climate such as humidity and seasons; breeding between existing trees and mutation.

Rendering method used in the project is a novel way of rendering trees, suitable for procedural generation and real time rendering of animations. The rendering method can be used to render forests and different types of trees, given that they can be represented as a branching structure. Compared to triangle meshes, parametric rendering used in this project takes up less memory and disc space, more suitable for real time changes to the structure of the tree, and real time animation. The proposed method is suitable to be improved for use in massive scales with further optimizations. Such optimizations may include; different level-of-detail methods that could be applied on branch, tree or forest level; different methods of parametric rendering such as ray tracing.

Developed user application is a proof-of-concept application used for displaying the results of implemented techniques for tree generation and rendering, and it has a easy to use and powerful UI. The application is suitable for development to become a full fledged modelling tool that can be used to generate trees. Generation for different structures of natural landscapes could also be included in the application in the future, which could be used in games or animation, though this project only focuses on trees. The application can be integrated to existing game engines or modelling programs for further ease-of-use.

References

[1] 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, jul 2019.

[2] W. Palubicki, K. Horel, S. Longay, A. Runions, B. Lane, R. Měch, and P. Prusinkiewicz, "Self-organizing tree models for image synthesis," in *ACM SIGGRAPH 2009 papers*, SIGGRAPH '09, (New York, NY, USA), p. 1–10, Association for Computing Machinery, 7 2009.

[3] 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, 8 1996.

[4] A. Runions, B. Lane, and P. Prusinkiewicz, "Modeling trees with a space colonization algorithm," in *Proceedings of the Third Eurographics Conference on Natural Phenomena*, NPH'07, (Goslar, DEU), p. 63–70, Eurographics Association, 2007.

[5] O. Deussen, P. Hanrahan, B. Lintermann, R. Měch, M. Pharr, and P. Prusinkiewicz, "Realistic modeling and rendering of plant ecosystems," in *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, (New York, NY, USA), p. 275–286, Association for Computing Machinery, 7 1998.

[6] P. Prusinkiewicz, L. Mündermann, R. Karwowski, and B. Lane, "The use of positional information in the modeling of plants," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, (New York, NY, USA), p. 289–300, Association for Computing Machinery, 8 2001.

[7] O. Stava, S. Pirk, J. Kratt, B. Chen, R. Mźch, O. Deussen, and B. Benes, "Inverse procedural modelling of trees," *Comput. Graph. Forum*, vol. 33, p. 118–131, sep 2014.

[8] A. Jakulin, "Interactive vegetation rendering with slicing and blending," 09 2000.

[9] L. Yi, H. Li, J. Guo, O. Deussen, and X. Zhang, "Light-guided tree modeling of diverse biomorphs," 2015.

[10] 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.

[11] R. Borchert and N. Slade, "Bifurcation ratios and the adaptive geometry of trees," *Botanical Gazette*, vol. 142, 09 1981.

[12] P. de Reffye, C. Edelin, J. Françon, M. Jaeger, and C. Puech, "Plant models faithful to botanical structure and development," *SIGGRAPH Comput. Graph.*, vol. 22, p. 151–158, jun 1988.

[13] D. Barthélémy and Y. Caraglio, "Plant architecture: A dynamic, multilevel and comprehensive approach to plant form, structure and ontogeny," *Annals of botany*, vol. 99, pp. 375–407, 04 2007.

[14] R. Barringer, C. J. Gribel, and T. Akenine-Möller, "High-quality curve rendering using line sampled visibility," p. 1–10, 11 2012.

[15] Kohek and D. Strnad, "Interactive synthesis of self-organizing tree models on the gpu," *Computing*, vol. 97, pp. 145–169, 02 2014.

[16] A. Takenaka, "A simulation model of tree architecture development based on growth response to local light environment," *Journal of Plant Research*, vol. 107, pp. 321–330, 1994.

[17] L. Yi, H. Li, J. Guo, O. Deussen, and X. Zhang, "Tree growth modelling constrained by growth equations: Tree growth modelling," *Computer Graphics Forum*, vol. 37, 08 2017.

[18] B. Tóth and S. Szenasi, "Tree growth simulation based on ray-traced lights modelling," *Acta Polytechnica Hungarica*, vol. 17, pp. 221–237, 01 2020.

[19] B. Beneš, N. Andrysco, and O. Šťava, "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.

[20] B. Benes, "An efficient estimation of light in simulation of plant development," in *Computer Animation and Simulation '96* (R. Boulic and G. Hégron, eds.), pp. 153–165, Springer Vienna, 1996.

[21] A. Reshetov and D. Luebke, "Phantom ray-hair intersector," *Proc. ACM Comput. Graph. Interact. Tech.*, vol. 1, aug 2018.

[22] 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," *ACM Trans. Graph.*, vol. 31, pp. 50:1–50:10, July 2012.

[23] Y. Livny, S. Pirk, Z. Cheng, F. Yan, O. Deussen, D. Cohen-Or, and B. Chen, "Texture-lobes for tree modelling," in *ACM SIGGRAPH 2011 Papers*, SIGGRAPH '11, (New York, NY, USA), Association for Computing Machinery, 2011.

[24] S. Pirk, B. Benes, T. Ijiri, Y. Li, O. Deussen, B. Chen, and R. Měch, "Modeling plant life in computer graphics," in *ACM SIGGRAPH 2016 Courses*, SIGGRAPH '16, (New York, NY, USA), Association for Computing Machinery, 2016.

[25] M. Han, I. Wald, W. Usher, Q. Wu, F. Wang, V. Pascucci, C. Hansen, and C. Johnson, "Ray tracing generalized tube primitives: Method and applications," *Computer Graphics Forum*, vol. 38, pp. 467–478, 06 2019.

[26] H. Nuić and Mihajlović, "Algorithms for procedural generation and display of trees," in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 230–235, 2019.

[27] J. Hart, B. Baker, and J. Michaelraj, "Structural simulation of tree growth and response," vol. 19, pp. 151–163, 05 2003.

[28] A. Shek, D. Lacewell, A. Selle, D. Teece, and T. Thompson, "Art-directing disney's tangled procedural trees," in *ACM SIGGRAPH 2010 Talks*, SIGGRAPH '10, (New York, NY, USA), Association for Computing Machinery, 2010.

[29] J. Weber and J. Penn, "Creation and rendering of realistic trees," in *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, (New York, NY, USA), p. 119–128, Association for Computing Machinery, 1995.

[30] H. Honda, "Description of the form of trees by the parameters of the tree-like body: effects of the branching angle and the branch length on the sample of the tree-like body.," *Journal of theoretical biology*, vol. 31 2, pp. 331–8, 1971.

[31] A. Lindenmayer, "Mathematical models for cellular interactions in development i. filaments with one-sided inputs," *Journal of Theoretical Biology*, vol. 18, no. 3, pp. 280–299, 1968.

[32] W. T. Reeves and R. Blau, "Approximate and probabilistic algorithms for shading and rendering structured particle systems," *SIGGRAPH Comput. Graph.*, vol. 19, p. 313–322, jul 1985.

[33] P. E. Oppenheimer, "Real time design and animation of fractal plants and trees," in *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, (New York, NY, USA), p. 55–64, Association for Computing Machinery, 1986.

[34] S. Ulam, *On some mathematical problems connected with patterns of growth in figures*, pp. 215–224. 1962.

[35] W. Palubicki, *Fuzzy Plant Modeling with OpenGL- Novel Approaches in Simulating Phototropism and Environmental Conditions*. Saarbrücken, DEU: VDM Verlag, 2007.

[36] J. C. Hart, D. J. Sandin, and L. H. Kauffman, "Ray tracing deterministic 3-d fractals," *SIGGRAPH Comput. Graph.*, vol. 23, p. 289–296, jul 1989.

[37] O. Cornut, "Dear imgui." https://github.com/ocornut/imgui.

[38] I. Quilez, "Quadratic bezier - distance." https://www.shadertoy.com/view/ldj3Wh, Dec 2013.

[39] I. Quilez, "Bezier bounding box." https://iquilezles.org/articles/bezierbbox/, 2018.

[40] The Khronos Group, "Shader storage buffer object." https://www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object, June 2020.

[41] J. F. Blinn, "Simulation of wrinkled surfaces," *SIGGRAPH Comput. Graph.*, vol. 12, p. 286–292, aug 1978.