CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Master's Thesis

Falling leaves simulation

MARTIN PAŽOUT

Supervisor: Ing. Jaroslav Sloup
Field of study: Open Informatics
Subfield: Computer Graphics
June 2022

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Pažout**	Jméno: **Martin**	Osobní číslo: **468953**
Fakulta/ústav:	**Fakulta elektrotechnická**		
Zadávající katedra/ústav:	**Katedra počítačové grafiky a interakce**		
Studijní program:	**Otevřená informatika**		
Specializace:	**Počítačová grafika**		

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Simulace padajícího listí

Název diplomové práce anglicky:

Falling leaves simulation

Pokyny pro vypracování:

Prostudujte metody simulace padajícího listí [1-7] používané v počítačové grafice. Na základě prostudované literatury vytvořte interaktivní aplikaci, která bude simulovat v reálném čase padání a pohyb listů ve větru. Zaměřte se na generování dostatečného množství tvarově a barevně odlišných listů, aby simulace vypadala věrohodně. Uživatelské rozhraní aplikace rozšiřte o možnost změny všech parametrů ovlivňujících vlastní simulaci včetně změny směru i rychlosti větru. Vygenerované listy alespoň tří druhů stromů porovnejte s reálnými fotografiemi. Vyhodnoťte rychlost implementované metody a její paměťovou složitost.
Implementaci proveďte v C/C++ s využitím OpenGL, CUDA/OpenCL.

Seznam doporučené literatury:

[1] Daeyeoul Kim, Jinmo Kim: Procedural Modeling and Visualization of Multiple Leaves. Multimedia Systems, vol.23, no.4, p.435-449, Springer, 2017.
[2] Yinling Qian et al. : Layered Leaf Texturing Using Structure-guided Model. Graphical Models, Vol.103, May 2019, Article 101029, 2019.
[3] SoHyeon Jeong, Si-Hyung Park, Chang-Hun Kim: Simulation of Morphology Changes in Drying Leaves. Computer Graphics Forum, vol.32, no.1, p.204-215, Blackwell Publishing Ltd, 2013.
[4] Xiaomin Wang, Chunjiang Zhao, Shenglian Lu, Xinyu Guo: Survey on Modeling and Visualization of Plant Leaf Color. Third International Symposium on Plant Growth Modeling, Simulation, Visualization and Applications, pp. 417-424, 2009.
[5] Ying Tang, Dong-Yan Wu, Jing Fan: Computational Approach to Seasonal Changes of Living Leaves. Computational and Mathematical Methods in Medicine, vol.2013, Article ID 619385, 2013.
[6] Chengyang Li et al.: GPU Based Real-time Simulation of Massive Falling Leaves. Computational Visual Media, vol.1, no.4, p.351-358, Springer, 2015.
[7] Ed Quigley et al.: Real-time Interactive Tree Animation. IEEE Transactions on Visualization and Computer Graphics, vol.24, no.5, p.1717-1727, 2017.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Jaroslav Sloup **Katedra počítačové grafiky a interakce**

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **01.02.2022** Termín odevzdání diplomové práce: **20.05.2022**

Platnost zadání diplomové práce: **30.09.2023**

Ing. Jaroslav Sloup podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

Acknowledgement / Declaration

I would like to thank my thesis supervisor Ing. Jaroslav Sloup for managing me and helping me throughout the whole process. I would also like to thank prof. Ing. Daniel Sýkora for help with texture part of the thesis.

I declare that this thesis represents my work and that I have listed all the literature used in the bibliography.

Prague, 20. 5. 2022

Abstrakt / Abstract

Tato práce se zabývá simulací padajícího listí. Prvním cílem práce je vygenerovat dostatečné množství tvarově a barveně odlišných listů. Druhým cílem práce je simulovat pád listu v reálném čase použitím GPU. Nasledně je vytvořena aplikace pro simulaci pádu listu, ve které lze změnit parametry.

Klíčová slova: Grafika, Simulace, C++, OpenGL, CUDA, OpenCV, Generovaní textur

Překlad titulu: Simulace padajícího listí

This thesis deals with simulation of falling leaves. The first goal of the thesis is to generate sufficient amount of leaves with distinct color and shape. The second goal of the thesis is simulate leaf fall in real time using GPU. An application is created for this simulation, where user can set the parameters.

Keywords: Graphics, Simulation, C++, OpenGL, CUDA, OpenCV, Texture generation

Contents /

1 Introduction **1**
1.1 Goals 1
1.2 Thesis Structure 1

2 Analysis **2**
2.1 Related work 2
2.1.1 Survey on Modeling and Visualization of Plant Leaf Color 2
2.1.2 Procedural modeling and visualization of multiple leaves 2
2.1.3 Layered leaf texturing using structure-guided model 2
2.1.4 Simulation of Morphology Changes in Drying Leaves 3
2.1.5 Plant leaves visualization based on leaf vein extraction 3
2.1.6 Realistic Simulation of Seasonal Variant Maples . . . 3
2.1.7 Computational Approach to Seasonal Changes of Living Leaves . . 3
2.1.8 Blowing in the Wind 3
2.1.9 GPU based real-time simulation of massive falling leaves 3
2.1.10 Modeling Autumn Sceneries . 3
2.1.11 Real-time Interactive Tree Animation 3
2.2 Leaf acquisition 5
2.2.1 Mask 5
2.2.2 Vein generation 5
2.2.3 Color 5
2.3 Simulation 5

3 Leaf Acquisition **7**
3.1 Contour Generation 7
3.1.1 Input 7
3.1.2 Pixel Contour 7
3.1.3 Simple Filter 7
3.1.4 Ramer-Douglas-Peucker filter 9
3.2 Vein growth 9
3.2.1 Main Vein 10
3.2.2 Find START and TARGET for main vein . . 11
3.2.3 Grow Vein 12
3.2.4 Vein Curve 12
3.2.5 Lateral Vein 13
3.2.6 Find START and TARGET for lateral vein . 13
3.2.7 Result 15
3.3 Texture acquisition 16
3.3.1 Input 16
3.3.2 Structural Distance 17
3.3.3 Region Creation 17
3.3.4 Superpixels 20
3.3.5 Region Selection 21
3.3.6 Pyramid Creation 23
3.3.7 Superpixel Selection 24
3.3.8 Image smoothing 24

4 Tree Simulation **27**
4.1 Wind Field Generation 27
4.2 Skinning 27
4.3 Tree Data 29
4.4 Tree Simulation 34
4.5 Original idea for implementation 35
4.6 Tree simulation 38
4.6.1 Dynamics calculation . . . 38
4.6.2 Integration of movements . 42
4.6.3 Results 43

5 Leaf Falling simulation **45**
5.1 Motion modeling 45
5.1.1 Trajectory generation . . . 45
5.1.2 Rotation 50
5.2 Low Dimensional Fall Trajectory Representation 52
5.2.1 Load Motions From Disk . 52
5.2.2 Create Transitional Probability Table 53
5.2.3 Create Trajectories D . . . 53
5.3 Leaf Class 54
5.3.1 Leaf Generation 55
5.4 Instanced rendering 55
5.5 Render 58
5.5.1 Prepare motions and Ds for CUDA 58

5.5.2 Leaf Update 59
5.5.3 Shaders 61
5.6 GUI 63

6 Results **64**
6.1 Leaf Acquisition 64
6.2 Tree Simulation 66
6.3 Leaf Falling Simulation 66

7 Conclusion **70**

References **72**

Tables / Figures

6.1 Falling Simulation Speed 67

2.1 Compared methods 4
3.1 Input Image 8
3.2 Pixel Contour 8
3.3 Simple Filter Code 9
3.4 Douglas Peucker algorithm 10
3.5 Contour Levels 10
3.6 Main vein growth 11
3.7 Main vein growth algorithm ... 11
3.8 Principle of Vein Grow Algorithm.......................... 12
3.9 De Casteljau algorithm 13
3.10 Lateral Vein Growth Algorithm.......................... 14
3.11 Lateral Vein Growth 14
3.12 Vein Generation Result 15
3.13 Example input images 16
3.14 Target input images............ 17
3.15 Structural Distance BST 18
3.16 Structural Distance Image 19
3.17 Leaf Region class 20
3.18 Regions 20
3.19 Leaf Superpixel struct 21
3.20 Leaf Superpixel Generation.... 22
3.21 Superpixel result 22
3.22 Region Selection................ 23
3.24 Pyramid Level structure 23
3.23 Pyramid Levels................. 24
3.25 Selection progression........... 24
3.26 Smoothing 26
3.27 Smoothing Detail 26
4.1 Wind Simulation Visualistion . 27
4.2 Bone Weight Demonstration... 29
4.3 Bone Structure Example....... 30
4.4 Skinning Vertex Shader........ 30
4.5 Blender Tree Model 31
4.6 Blender Atlas of Primitives 32
4.7 Rigid Body Constant Data 32
4.8 Rigid Body Changing Data.... 33
4.9 Tree Cylinder Structure........ 33
4.10 Tree Flatten Example.......... 35
4.11 Tree rigid body initialization .. 36
4.12 Illustration of tree dynamics... 37
4.13 Tree simulation overview 39
4.14 Propagate rotation illustration 42
4.15 Tree Simulation Result 43

5.1 Primitive Motions 46
5.2 Steady Descend 47
5.3 PT, TC and PF Trajectory Motions 48
5.4 Top View of PS and TH 48
5.5 PS and TH trajectory motions . 49
5.6 Schematic ilustration of the forces on the leaf 50
5.7 PT,TC and PF ODE Call Function 51
5.8 Angular velocity visualisation . 52
5.9 Transition Probability Table . . 53
5.10 Trajectory Feature D 54
5.11 Leaf Class 55
5.12 Leaf Generation 55
5.13 Instancing Example 56
5.14 Instanced Attribute Set 57
5.15 Leaf render function 57
5.16 Leaf Render Overview 58
5.17 Meta class 59
5.18 Sample D ALG 60
5.19 Sample Motion ALG 60
5.20 Apply Change Code 61
5.21 Change Functions 61
5.22 Vertex Shader 62
5.23 Fragment Shader 62
5.24 Graphical User Interface 63
6.1 Leaf Vein Generation Output . 64
6.2 Leaf Texture Comparison 65
?? Tree Simulation Result . . . . ??
6.4 Computed Trajectory Result . . 67
6.5 Fall Simulation Result 68
6.6 CUDA time measure 69

Chapter 1
Introduction

Leaves are important part of a tree and simulation of them falling adds a nice detail into the scene. Leaf are sort of an after thought of our mind that we don't really pay a close attention to their trajectory. Where if I asked you how leaves fall down you would probably have a hard time imagining it. Nonetheless they are important and add extra detail. Trees are also important part of any scenery and it would be weird if the leaves reacted to the wind but the tree would not so part of my thesis will also focus on how to make the tree move, make them more interactive. Also since there is not a lot of leaf texture images, part of my thesis also focuses them. The swaying of the trees will depend on user defined attributes and will run in real time.

1.1 Goals

There are two main outputs of this thesis. It will be one offline application to generate leaf textures. Some user input will be required so it will be semiautomatic. Ideally it would be automatic fully but due to complexity and automation error its more desirable to do it in semiautomatic manner. In this application I also don't focus on performance so this part will not be measured.

The second output will be realtime simulation of a scene. The scene will consists of several trees that react to accordingly to environmental influence. The environmental influence will consist of wind. Wind depending on various attributes will sway the trees. Leaves falling trajectory will also be affected by wind.

1.2 Thesis Structure

The thesis is structured as follows: chapter 2 consist of analysis of what approaches are available and creates a comparison between them. Chapter 3 will focus on acquiring the leaf texture. Chapter 4 will discuss the tree simulation. Chapter 5 is about the Leaf Falling simulation. Chapter 6 will present the results of the work. And chapter 7 is the conclusion.

Chapter 2
Analysis

This chapter focuses on analysis of the problem. The main interest of this section is to find and compare available research in related to leaf acquisition and the whole simulation of the tree and its leaves.

As mentioned before the whole application will consist of two applications. The first will be an offline application handling leaf model creation. The second one will be a simulation of a scene with trees that react to the environment changes. The solution will combine the publications from related work. To be precise we will use the first application to generate leaf various textures that the second application will use.

2.1 Related work

A lot of research went into the simulation of leaves and trees. Namely, a Survey on Modeling and Visualization of Plant Leaf Color [1], discusses possible ways on how to visualize a leaf. A lot of methods focus on one main thing and simplify other input data. Listed below are relevant papers and what methods are relevant to my application. The relevant papers are ordered by when I will use them in implementation process. Overall overview of used methods is in Table 1.

2.1.1 Survey on Modeling and Visualization of Plant Leaf Color

This paper [1] provides a survey of research on the modeling and visualization of plant leaf color based on computer graphics and physiology. I used this as reference point for possible implementation methods.

2.1.2 Procedural modeling and visualization of multiple leaves

Study [2] describing method to model various type of leaf from mask of an image. Contour of the leaf is approximated using Douglas-Peucker algorithm. Method further grows veins of the leaf based on contour information. Lastly it describe a way how to generate color of the leaf. This is the main paper that I am implementing in this semester project.

2.1.3 Layered leaf texturing using structure-guided model

This publication [3] focuses on procedural texturing of a leaf based on its vein structure. It requires an atlas of reference leaf images and their respective vein structure. The image is divided based on the vein structure and pixels of the respective reference image are saved depending on distance from the root of the vein. The distance is computed based on how long would it take to supply the given pixel with water. With this method I plan to create textures for generated leaves from[2].

2.1.4 Simulation of Morphology Changes in Drying Leaves

Introduces [4] a biologically motivated simulation technique for the realistic shape deformation of drying leaves. This method simulates the whole leaf surface to capture the fine details of desiccated leaves. I potentially plan to use similar method to create further leaf variation.

2.1.5 Plant leaves visualization based on leaf vein extraction

Paper [5] about automatic reconstruction of veins from leaf images. It uses HSI color space to extract radiance component based on which it determines the vein. This paper further describes triangulation and deformation of the resulting leaf.

2.1.6 Realistic Simulation of Seasonal Variant Maples

This publication [6] presents a biologically motivated system of seasonal change of leafs. It is based on amount of cell that make up the color of the leaf. This method describes how the coloring of a leaf works in biological sense and I plant to use it as theoretical background to create relationships in Markov chains in [7].

2.1.7 Computational Approach to Seasonal Changes of Living Leaves

This paper [7] proposes a computational approach to seasonal changes of living leaves by combining the geometric deformations and textural color changes. From this paper I plan to use same approach to model change of leafs according to seasons. The method uses Markov chains to describe states of a leaf. The move between states is based on temperature and wetness of the environment the leaf is in.

2.1.8 Blowing in the Wind

Publication [8] is about simulating fall of a feather and bubble. Models flow field using Lattice Boltzmann method (LBM) to simulate wind. This paper approximates feather with bezier curve and applies the flow filed forces on it. In future part of this work I plan to use similar approximation only instead of feather i will have leaf.

2.1.9 GPU based real-time simulation of massive falling leaves

Publication [9] is about simulation large number of falling leaves using the raw computation power of GPU. For approximation of the fall trajectory it uses motion synthesis based method to analyze potential falling trajectory which is then described in lower dimensions using predefined set of simple trajectories. Based on this method i plan to create these atlases of motions to similarly approximate falling trajectory.

2.1.10 Modeling Autumn Sceneries

Publication [10] describes aging of a leaf using atlas of discrete states similarly as in [9] but does not focus on GPU implementation mainly on method itself. Also includes collision approximation of leaf using simple disc primitive.

2.1.11 Real-time Interactive Tree Animation

Publication [11] is about simulation of tree using rigid bodies. The tree consist of large amount of rigid bodies on which force can be applied. The model of the tree it self is created using rigid body software. Based on this method I want to model tree and wind relationship.

Publication	Automatic texture extraction	Vein generation	Leaf triangulation	Leaf texture synthesis	Leaf fall trajectory	Tree environ-ment aware	Leaf variation
Blowing in the wind [20]					YES		
GPU based realtime simulation of massive falling leaves[11]					YES		
Computational Approach to seasonal changes [16]			YES				
Realistic Simulation of Seasonal Variant Maples [21]				YES			
Modeling autumn sceneries [3]			YES		YES		YES
Plant leaves visualisation based on leaf vein extraction [22]	YES	YES	YES				YES
Procedural modeling and visualization of multiple leaves [7]		YES	YES	YES			YES
Realtime interactive tree animation [15]					YES	YES	
Simulation of Morphology Changes in Drying Leaves [5]				YES			YES
Layered leaf texturing using structure-guided model [14]				YES			YES

Figure 2.1. Overall capabilities of methods described in publications.

2.2 Leaf acquisition

Since leaf texture models with a lot of variety are not widely available I have opted to use procedural generation of leaves. With procedural generation, one can create a variety of images in no time. A lot of publication focuses on leaf acquisition. So I chose to combine methods from related work to create the desired method. I chose a data-driven approach where from leaf images I generate more. Basically I will create an atlas of reference images that can be easily used.

2.2.1 Mask

The first thing I need is to mask out the background of the leaf image. Currently, for the ease, I opted to create mask manually. One might ask why I didn't choose fully automated approach and the reason is simple, because its simpler and will provide better results. I tested the automatic approach in Krita [12] that automatically distinquish background and the desired leaf but most of the time it didn't work. And if a professional software cannot create the desired result its very unlikely that mine would. There are some papers like [5] that focus on this topic but I will leave this as a future improvement.

2.2.2 Vein generation

With the mask of the image, I use methods described in [2]. First I will compute the contour of the mask of the image. I use simple neighbour algorithm to detect the pixel contour of the image first. Since in this countour there would be too many unnecessary pixels I use the Douglas Peucker algorithm [13] to simplify it.

This contour will be then used to guide vein growth. To simplify things we will grow two levels of veins: main veins and lateral veins. Where lateral veins will grow from main vein. The growth depends on the shape of the contour, where we will start the vein growth and in which direction we will go. Further details will be explained in chapter Leaf Acquisition.

2.2.3 Color

The color of the leaf is generated using methods from [3] With these methods we create a texture for the given shape of the leaf. It is a reference based approach, where we map color from existing image to a new image. This process depends on the structure of the leaf based on which we match patches of color from example image to the target one. The process is also randomized so we can produce various leaf textures with ease.

2.3 Simulation

The simulation take into account the given wind in the scene. But reaction of all objects in the scene would be too expensive. So not all objects in the scene will react to the incoming wind but only the trees and leaves. The wind is going to be approximated by a wind field and simple user input.

To apply wind force to leaves we will approximate it by a curve similarly as in [8]. For the trajectory of the fall, we use the method from [9]. We will create a range of possible ways of how the leaf could fall. According to the wind force and leaf position, we create a falling trajectory. Which is composed of the range of possible trajectory falls.

Trees will react to the wind by using rigid bodies. We create a tree model by using a special process in blender for rigid-body modeling. By using methods from [11] we will apply the wind force onto the tree. We will also test if enough force is applied to the leaf through the tree body and if we surpass the leaf-falling threshold we will start the leaf falling process.

Chapter 3
Leaf Acquisition

In this section I will describe implementation detail of leaf acquisition. The implementation focuses on contour creation and vein growth from [2] and on generation color texture as in [3]. I have combine methods from both of these paper to fit my needs. That is I needed a way to generate veins to get input for color texture and I needed color for the picture around the vein so these methods matched each other perfectly.

3.1 Contour Generation

Contour generation is heavily based on [2]. First we load image mask representing the whole area that the leaf is in. Than we find pixel contour of the mask and apply two filters that reduce the number of need points to represent contour greatly while maintaining overall shape. Contour itself is used latter in algorithm to create TARGET pixels for vein growth algorithms. In code all contour algorithms are wrapped in ContourHandler class.

3.1.1 Input

Input in contour generation is a PNG image that has only two colors BLACK and WHITE. If you use JPG images you will face a difficulty where JPG's have an automatic smoothing build inside them and the mask pixels will break. For example pixel with *(255,255,255) change to (128,128,128)* on the edges of the image. While it make sense to use bitmap image format i decided to use PNG for the ease of result image creation. Image class handles all loading and saving of images. It is a wrapper for stb library [14] for writing and loading images. For creation of mask itself I used Krita [12]. In figure 3.1 you can see example input image.

3.1.2 Pixel Contour

For pixel contour i used the fact that it must be inside leaf mask color and also one of its neighbours is not. In code this translate into test if the pixel is WHITE check 8 neighbouring pixels if on of them is BLACK. While in [15] more elegant approach is used I have chosen this approach for ease of implementation and it also works as needed preparation for DouglasPeucker algotrihm [16] where we need lines not points. During this contour aproximation we prepare just that.

3.1.3 Simple Filter

Simple filter that instead of pixels uses lines. In that sense we remove such pixels of the image that lie on a line. First we find some contour pixel by traversing image starting from top left corner to bottom right corner and checking for pixel contour color.Once we find the starting point we begin traversing along the pixel contour similarly as in [15]. We build a basic state machine that depending from which direction we came from chooses if the current pixel will contribute into line representation of pixel contour or

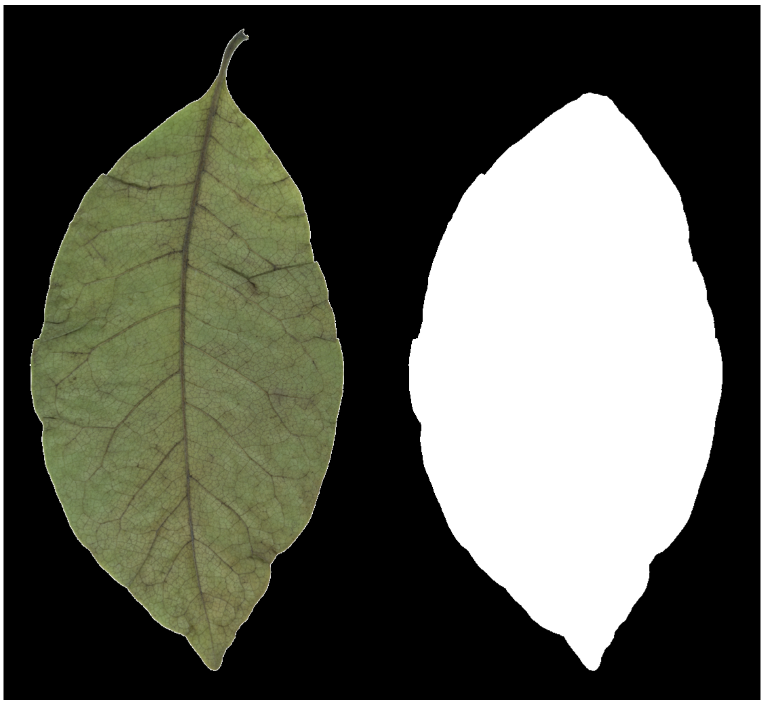


Figure 3.1. Left leaf texture, right leaf mask.

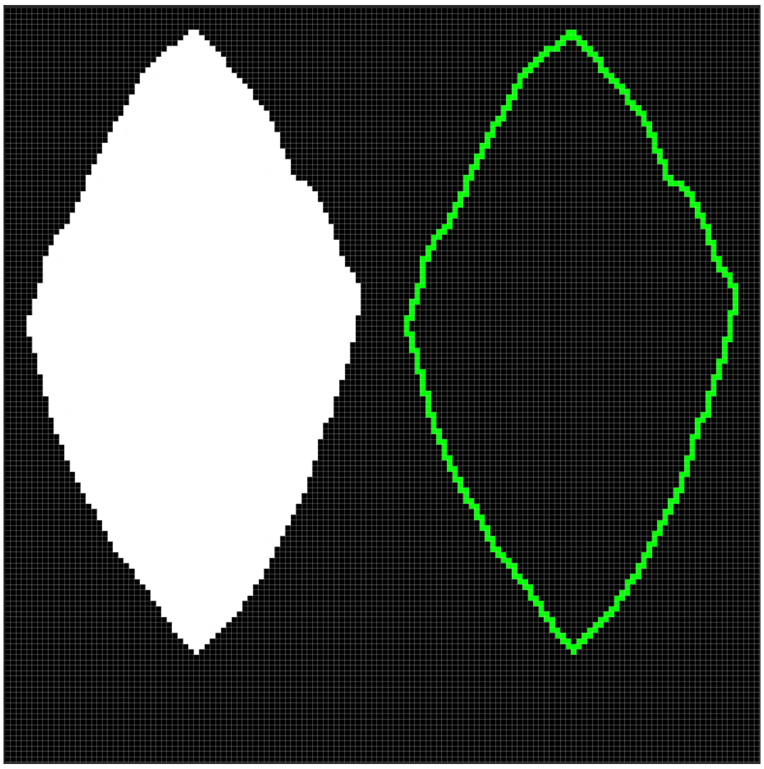


Figure 3.2. Left leaf texture, right leaf mask.

not. We go around the contour counterclockwise and add only pixels that have different direction. If they don't have a different direction they lie on a line. So once we detect a change of direction we add the start point and end point as a line. A one route of the state machine is described in figure 3.3. We come previously from the LEFT side and in the if statements we check if change of direction happened if so we save the line segment. Order of the if statements matter since we traverse in counterclockwise manner we need to order them in a way we traverse it as such.

```
if(Came from LEFT){
    if{ UP pixel is contour }{
        add current pixel as line vertex;
        move UP;
    }
    else if{ LEFT pixel is contour }{
        we came from same direction we can skip this pixel;
        move LEFT;
    }
    else if{DOWN pixel is contour }{
        add current pixel as line vertex;
        move DOWN;
    }
    else if{RIGHT pixel is contour }{
        add current pixel as line vertex;
        move RIGHT;
    }
}
```

Figure 3.3. Code for deciding if we add pixel as a line vertex if we came from the LEFT.

3.1.4 Ramer-Douglas-Peucker filter

To simplify the amount of points need to represent contour even further I use Ramer-Douglas-Peucker (RDP for short) algorithm [16]. I call this algorithm on vector of points in which each two points after each other form a line representing the leaf contour. RDP is a recursive algorithm which creates an imaginary line between the first and last point. Than it will find a pivot point a point that is perpendicularly furthest from the line. We than compare the pivot with user defined epsilon. If the distance is more epsilon than we recursively call on set of points divided by pivot. If the distance is less than epsilon we return only the start and end point. The recursion is described in figure 3.4. The order of reduction of the contour points depends on the chosen epsilon in figure 3.5. The higher the epsilon the more reduction.

3.2 Vein growth

This section focuses on the growth of the veins. It follows the process described in paper [2]. There are two levels of veins: the main vein and the lateral vein. The grow it self is the same form both main and lateral veins. What is different are the starting points, where the main vein starts at the root of the input image. That can be user defined or automatically detected. The lateral vein starts at a point on the main vein.

```
INPUT: A list of points

l = line between first and last point
while( i < points.size()){
    if( distance d from line l < dmax ){
        index = i;
        dmax = d;
    }
    if(dmax > epsilon){
        ret1 <- recirsively call on 0 to index points
        ret2 <- recursively call on index to end points
        return ret1 and ret2
    }
    else{
        return start and end point
    }
}
```

Figure 3.4. Ramer-Douglas-Peucker recursion.

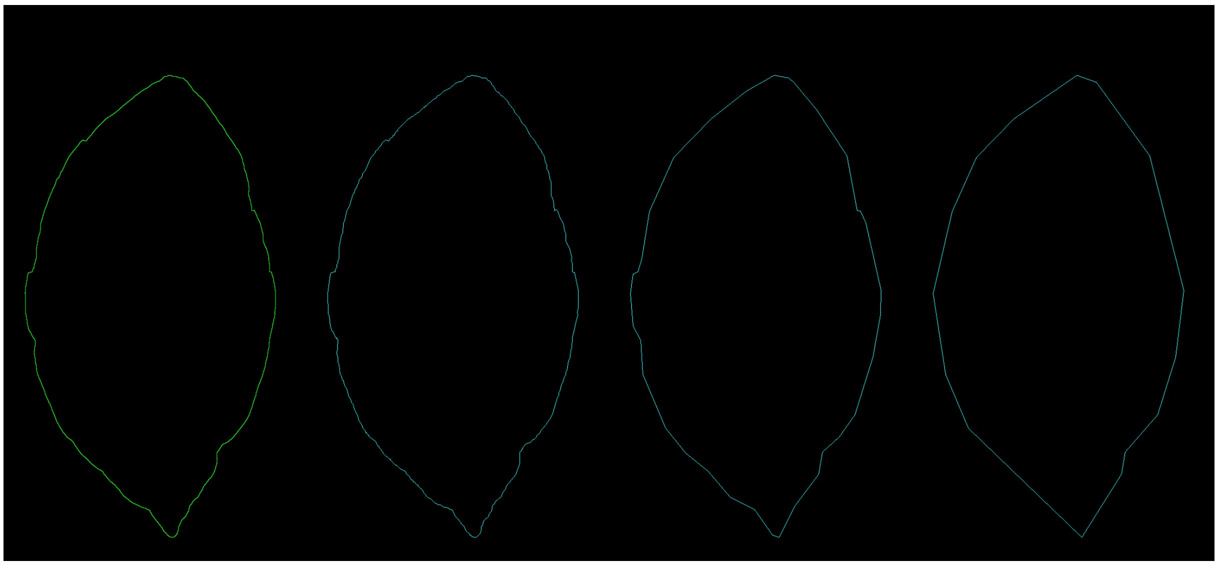


Figure 3.5. Result of contour approximation, from left to right respectively: pixel contour, 1.0f, 4.0f, 10.0f distance threshold.

3.2.1 Main Vein

The algorithm consist of 4 steps. First we find a START and TARGET. We need start and target to define the direction in which the vein should grow. Second step is to generate bezier control points between START and TARGET. Third we use the bezier control points in de Casteljau algorithm [17], to curve between these points. Lastly we connect these points with lines using bresenham algorithm [18]. All vein growing code is implemented in VeinGrowHandler class. Pseudocode for the algorithm is presented in figure 3.7 and the result is visualised in figure 3.6.

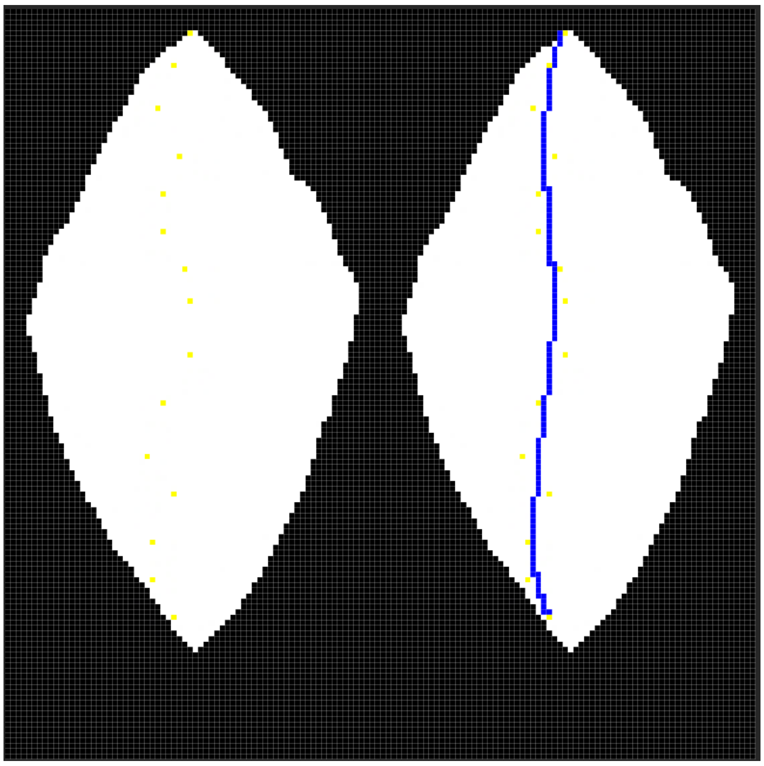


Figure 3.6. Left control points of bezier curve approximating main vein, Right resulting bezier curve.

```
INPUT: A list of contour points

start = find start point
target = find target point
while(pi distance not too close to target){
    vgr = target - pi
    rot = random angle [-30,30]
    scale = random angle [0.6,1.0]
    vi = scale * rot * vgr
    pi+1 = vi * step size
    add pi to vein control points
    pi = pi+1
}
visualize Bezire curve using DeCasteljau algorithm
```

Figure 3.7. Main vein growth algorithm.

3.2.2 Find START and TARGET for main vein

Similarly as in [2] I use a simple approach to find START and TARGET points. For START point we test all contour vertex points. For each of which we compute a distance from median x coordinate and distance from minimal y coordinate. And we simply take the one that has the lowest sum. For TARGET points we take the N furthest contour points from START that we found in previous step. The N indicates the number of main vein that we want to grow. It also should be noted that we use the filtered out contour points

3.2.3 Grow Vein

With START and TARGET points found, we can grow a vein as described in [2]. We create a direction vector from START to TARGET (simple TARGET - START). With a user-defined step variable, we multiple the direction vector. We randomly scale and rote this vector in user-defined 2D space (by scale and rotate variables). we move at the endpoint of the vector and start the process again. We continue this process until we get close enough to TARGET. The principle is illustrated on figure 3.8. Where p_i and v_i are the i-th chosen point and i-th direction vector. The vein is random but it is guided by the direction to the TARGET point.

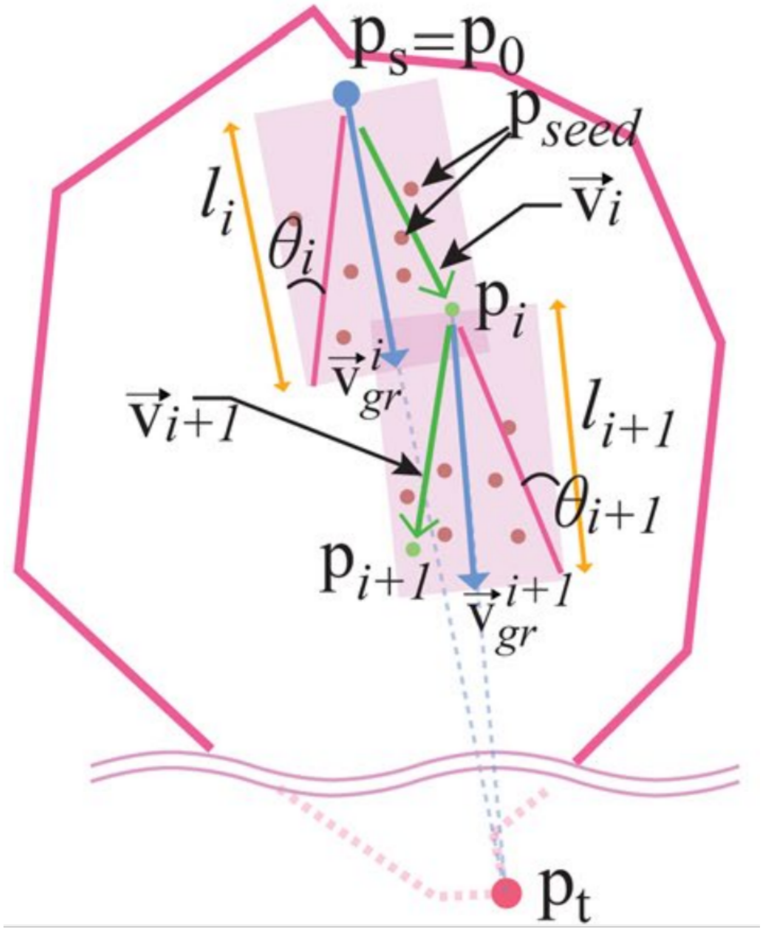


Figure 3.8. Principle of vein grow algorithm. Image taken from [2].

3.2.4 Vein Curve

The points created in the previous grow vein step are used as control points of Bezier curve. I used the De Casteljau algorithm [17] to compute the points on the curve. De Casteljau works as iterative Lerp function. In the first level it will compute lerp between all succeeding points and doing that we will get one less point than before. We than repeat this process untill we have only the last point left. Which is the on desired point on the curve. We do that in in the range $[0, 1]$ and take small step always. The

```
INPUT: A list of bezier control points

parentPoints = controlPoints //Copy control points
points(parentPoints.size() - 1)

int level = parentPoints.size();

//iterate until only one point remains
while (level > 1) {
    //iterate through all the points on the i-th level and save the Lerp
    //value for next iter
    for (int i = 1; i < level; i++) {
      p1 = parentPoints[i - 1];
      p2 = parentPoints[i];
      points[i - 1] = Lerp(p1, p2, a_t);
    }

    //make the lerp points new parent poinst and move to next level
    for (int i = 0; i < level - 1; i++) {
      parentPoints[i] = points[i];
    }

    level--;
  }
//the result value is the last interpolated value
return points[0];
```

Figure 3.9. De Casteljau algorithm.

quality or rather the amount points on the curve depends on a user-defined variable. The figure 3.9 shows the algorithm The created points are then connected by lines using the Bresenham algorithm [18].

3.2.5 Lateral Vein

Lateral vein growth depends on the main vein or rather the lateral veins start from points of the main curve. Also, the main vein works as a leaf divider, where space on the LEFT and RIGHT will have independent lateral veins. This will prove useful when testing for intersections between previous veins, since we dont need to test collisions between left and right side. The growth of a vein changes into finding START and TARGET points and growing a vein between them similarly as we grew the main vein. The START is chosen simply by taking a pixel step on the main curve. Potential TARGETS are contour points. TARGETS are chosen based on the angle they hold between the START and the point. The result is in figure 3.11. Pseudocode for the algorithm is presented in 3.10.

3.2.6 Find START and TARGET for lateral vein

To find a START we iterate over the points on the parent vein (main vein for lateral veins). We do not need to test every point so the variable step is defined to skip some

```
INPUT: A list of contour points
       A list of main vein points

left = get points left of main vein from contour points
i = 2 * step
while(i < main.size() - step){
    curr = i-th point on main
    maindir = approximate main direction for curr
    while(left points not iterated over){
        target = current contour point from left
        targetdir = target - curr
        angle = dot(maindir, targetdir)
        if(angle > angle treshold && no intersection between lateral veins){
            Grow lateral vein similarly as the main vein
        }
    }
    i = i + step
}
```

Figure 3.10. Lateral Vein Growth Algorithm of the left side of the leaf.

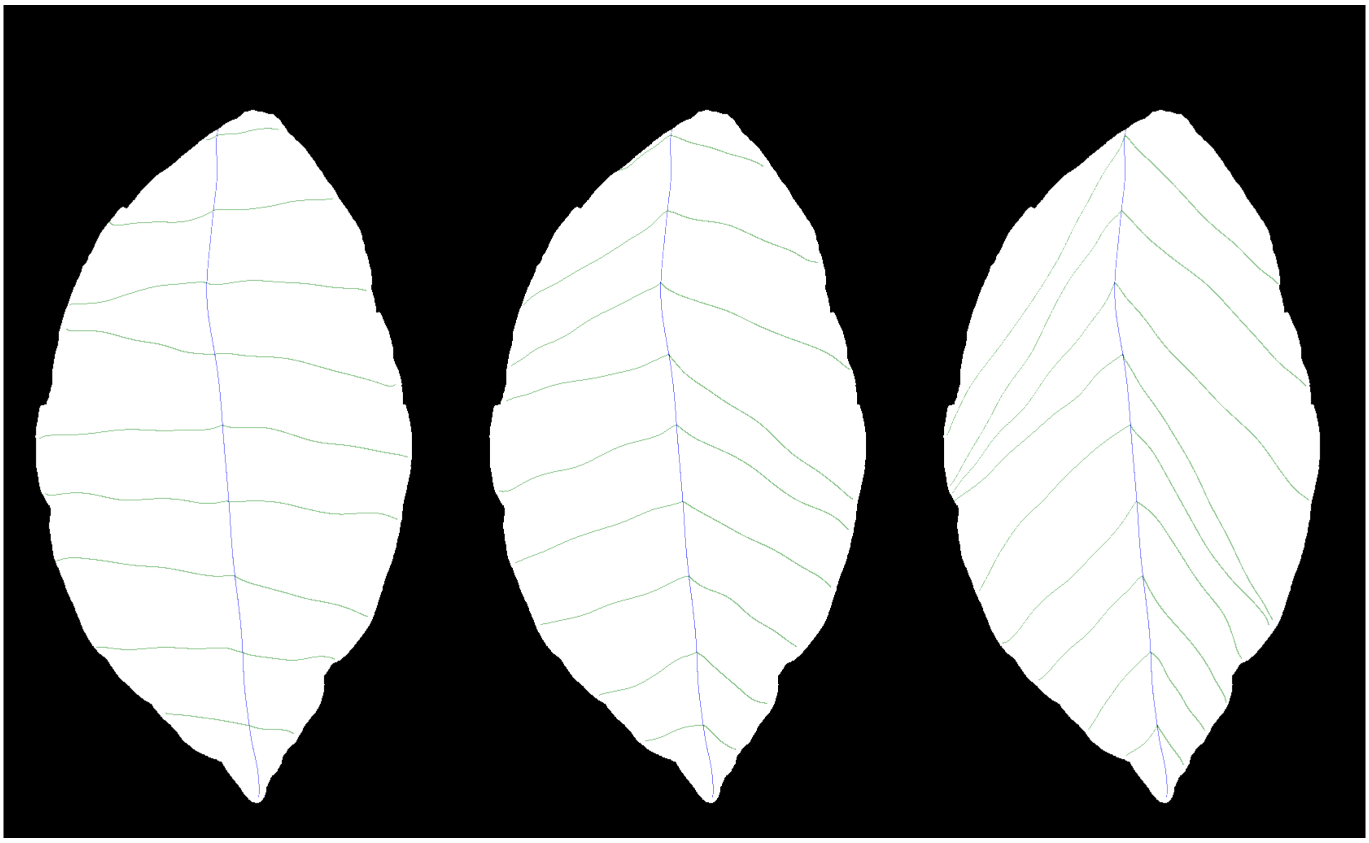


Figure 3.11. Lateral veins depending on angle threshold. From left to right angle = 90, 115, 140 respectively.

indexes. We also add start and end offset and a threshold for minimal index distance between previously created veins.

TARGET are points that define the contour of the leaf. If TARGET point is valid for a given START points depends on an angle a. First, we compute the direction cd of the current START and TARGET. Then we compute the angle a between cd and parent (main vein here) direction. If a is greather than user-defined variable we grew the vein. We also check for collision with previous lateral vein and skip this candidate if there is any.

3.2.7 Result

Combining all the mentioned algorithms for vein generation we will get and randomly generated vein structure as shown in figure 3.12. This generation is semi automatic and gives us the ability to generate various vein structures for further parts of the thesis. As you can see in the comparison (figure 3.12) the vein structure doesn't match that well to the original image, but we have to keep in mind that he image was used as an example generation not to be replicated but to be used for replication. In that in mind I am confident that more of these kind of result will be sufficient. What also should be mentioned that the vein structure only depends on the shape of the leaf which we can easily transform using image editing software like Krita [12] with color it would not be that easy.

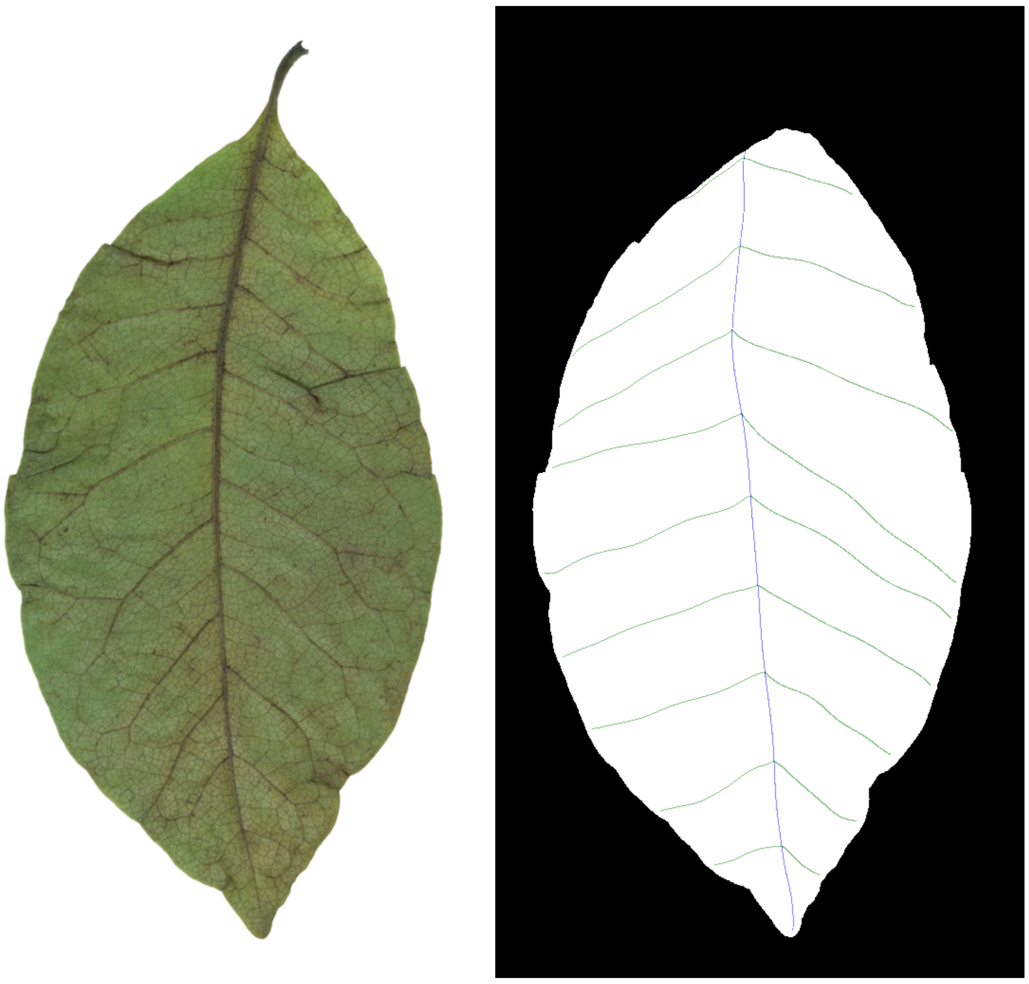


Figure 3.12. Comparison between reference texture and generated vein structure.

3.3 Texture acquisition

This section will describe the process of acquiring a color texture that will be used for the leaves. It follows the methods described in paper [3]. I have also opted to use known open source computer vision and machine learning software library OpenCV [19]. OpenCV has various image related algorithms and is widely use in the industry. It also eases the burden of implementing all of these algorithms by myself as it would prove difficult and time consuming to implement and debug.

3.3.1 Input

There are several input files that are need for the used methods. It should be also mentioned that there are two main kinds of input images. Example images that represent atlas of leaf textures as a source for generation. And target images that will based on example images be the output of the algorithm. I should also note since the algorithm uses randomness a image can be example image as well as a input image, but not the other way around. That is because the main difference between example and target image is that target doesn't necessarily need to have a color but example does.

For the convenience I will be using same picture for target and example. There are three input files for example image img_color, img_mask, img_structure that you can see in figure 3.13. All of these are prepared by hand in Krita [12].

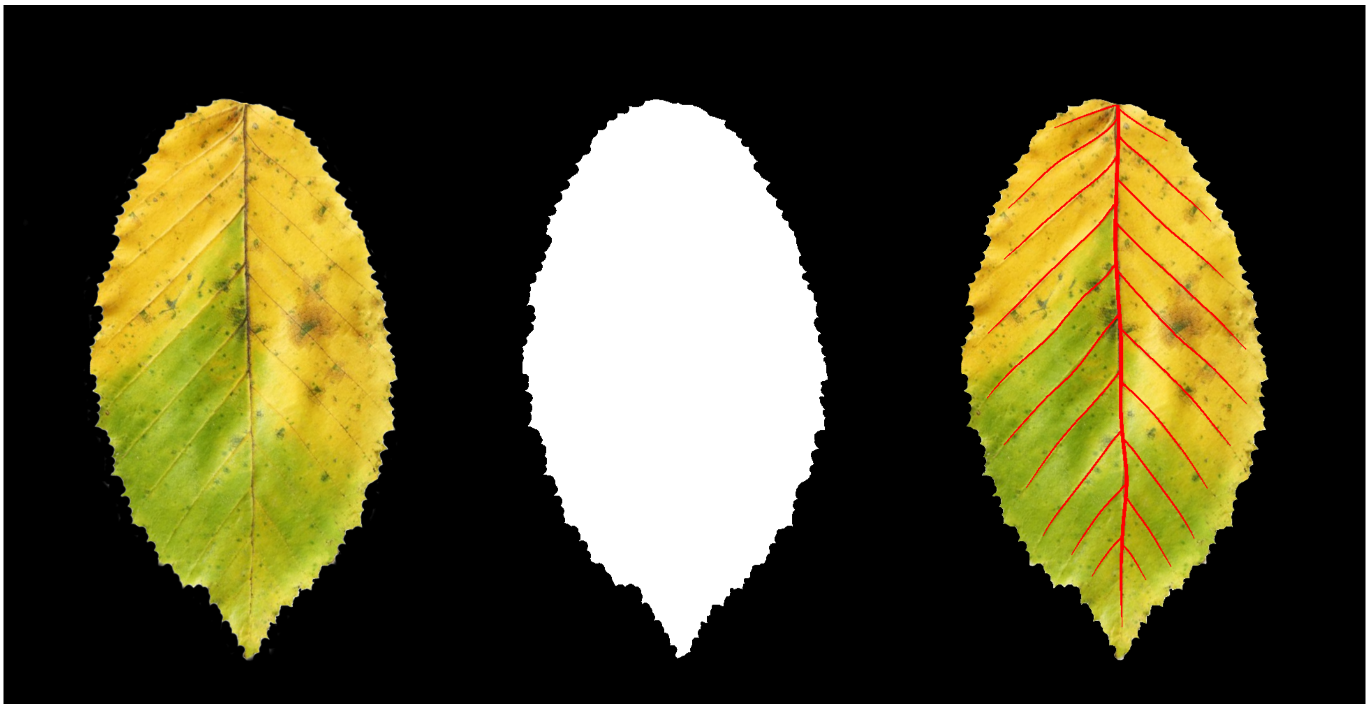


Figure 3.13. Example input images. Color image on the left, image mask in the middle and color with vein structure on the right.

As for the the target image its pretty similar only the img_color is same as img_mask. These can of course also be created manually, but the idea is to use the previously described vein generation algorithm paragraph 3.1 and 3.2. I have used both approaches. You can see an example of target input images in figure 3.14

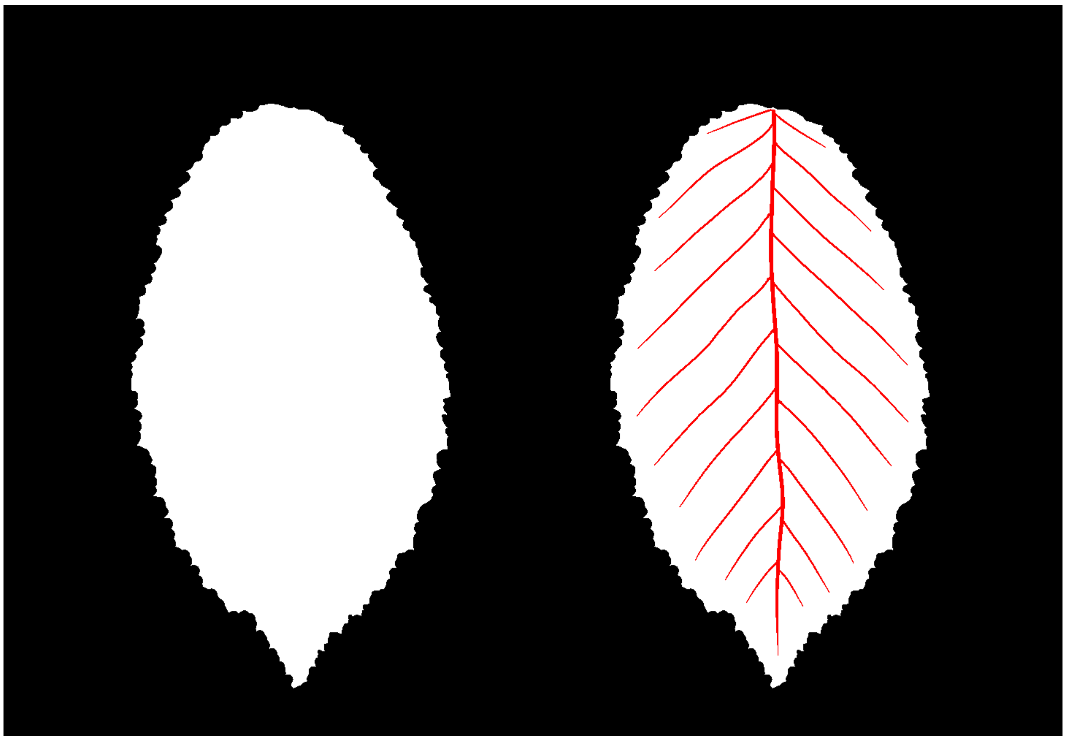


Figure 3.14. Target input images. Color/mask on the left side and color with vein structure on the right.

3.3.2 Structural Distance

For a similarity metric Structural Distance field is used in selection step of the algorithm to match similar patches of the color onto the new target texture. The structural distance is based on how would water travel through the leaf to each part of the leaf. There are three main water transfer states that can occur: VeinToVein, VeinToTissue, TissueToTissue. Technically there could be a fourth case with TissueToNothing but nothing happens there so its not further considered. The three transfer states define how hard is to transport water throughout the leaf. All we need is a point to start at and then we can traverse all the pixels and assign the best distance from the start point.

To chose a starting point a window will pop up where you can click into the image and set the starting point. This could have been done automatically in a similar way as in the contour generation. But this half automation will provide more precise results and is just better. A BST is then employed to to set the given values based on the transfer states. The algorithm is described in figure 3.15. In the figure 3.16 I have visualised how the structural distance look in greyscale, the darker it is the closer it is to the source position and you can clearly see how near the veins the structure is darker and in the tissues it is brighter.

3.3.3 Region Creation

Region creation is also done half automatic and half manually. To be precise a window will pop up of the color image with vein structure. User than has to mark start regions. These region are then used as an input for watershed algorithm [20]. Which will then create as many region as there are markers. Watershed algorithm is already

```
INPUT: A list of contour points
       A list of main vein points

StructuralDistance[startVein.x][startVein.y] = 0;
currRound.emplace_back(startVein);
while (!currRound.empty()) {
    for (auto p : currRound) {
        int myVal = StructuralDistance[p.x][p.y];
        CellType myType = IsWhatCellType(img, p.x, p.y);
        //check the neighbours
        for (int i = -1; i <= 1; i++) {
            for (int j = -1; j <= 1; j++) {
              if (i == 0 && j == 0) //skip itself
                continue;
              if (img_mask.at<cv::Vec3b>(p.x + i, p.y + j)
                    != cv::Vec3b(255, 255, 255))
                continue;

              CellType neighbourType = IsWhatCellType(img, p.x + i,
                                                     p.y + j);
              int neighbourVal = StructuralDistance[p.x + i][p.y + j];
              int potencialDistance = myVal;
              if (myType == CellType::VEIN &&
                    neighbourType == CellType::VEIN) {
                potencialDistance += VeinToVein;
              }
              else if (myType == CellType::VEIN &&
                          neighbourType == CellType::TISSUE) {
                potencialDistance += VeinToTissue;
              }
              else{
                potencialDistance += TissueToTissue;
              }
              if (potencialDistance < neighbourVal) {
                StructuralDistance[p.x + i][p.y + j] = potencialDistance;
                nextRound.emplace_back(p.x + i, p.y + j);
              }
            }
          }
        }
        currRound = nextRound;
        nextRound.clear();
      }
    }
}
```

Figure 3.15. Structural Distance BST.

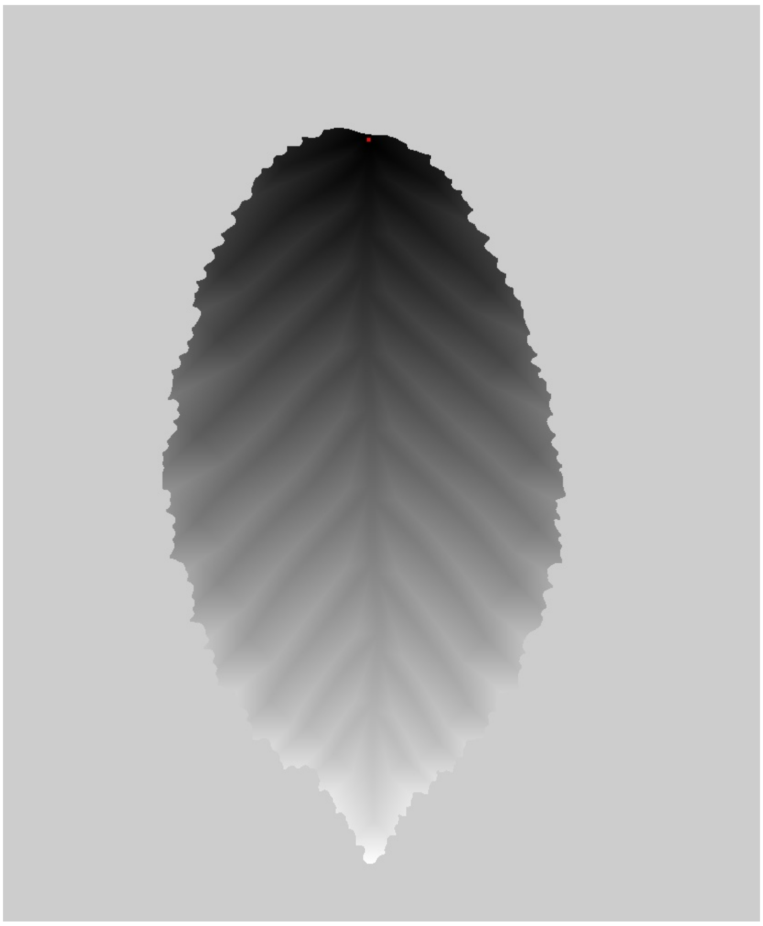


Figure 3.16. Structural Distance Image visualisation.

implemented in OpenCV [19] so I have used their implementation and created regions this way.

The OpenCV watershed algorithm outputs an matrix where in each cell there is a ID representing what region does this pixel belong to. Since regions are needed to be compared to each other and this representation is not desirable. Also additional information are desired to further help simplify and speed up the comparison. The comparison is follows the equation (1), where the IoU is an intersection over union and a_s and a_t is the area of the region. Area of the region is here approximated by number of pixels. This comparison will give higher similarity to region that are similar in size and similarly distant from the source of the Structure Distance.

$$p = \frac{IoU(r_s, r_t)}{1 + (a_s - a_t)^2} \tag{1}$$

Following the outlined requirement to compare two regions between each other I have created a class that hold the necessary information. The class ***LeafRegion*** is defined in figure 3.17. We want a minimal and maximal distance of the region, to get the distance range that the region covers. And pixel count to get the area of the region. The Leaf region also contains another class called region which simply holds the points and id of the region.

With this class prepared all we need to do is fill it with data. First allocate vector of the class LeafRegion with the size of the number of regions. Then we iterate over

```
class LeafRegion {
public:
  Region region;
  int minDist;
  int maxDist;
  int pixelCount;
};
```

Figure 3.17. Leaf Region class.

all of the IDs from the watershed output and put points in regions. We are using the ID to index the vector and access the desired region. We also skip any pixels that are not on the mask. Using that we got all the points that are in the regions all we need to do now is iterate over them and compute the minimal, maximal distances and pixel count and we have the regions prepared for selection that will be discussed below. The region pixels are visalised in figure 3.18.

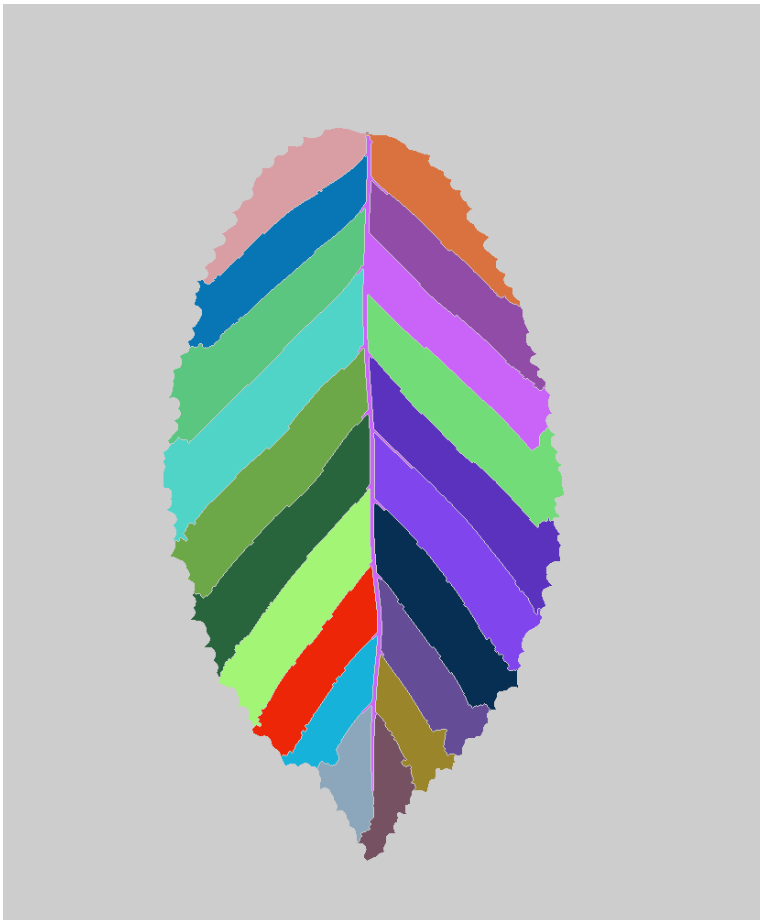


Figure 3.18. Regions visualisation.

3.3.4 Superpixels

Matching new texture based on leaf regions alone wouldn't be enough to provide sufficient texture quality. In addition leaf pixels are also group into superpixels. As in paper [3] the superpixels algorithm is not called on each region separately but on the leaf as a whole. Each superpixel is save in a struct ***LeafSuperpixel***. Similarly to regions additional information is save in this structure: a mean color and mean distance. It also contains the class region that holds points where the superpixel is. Additional information is useful in selection part of the application. You can see the LeafSuperpixel structure in figure 3.19.

```
struct LeafSuperpixel {
  cv::Vec3b color;
  int dist;
  Region region;
};
```

Figure 3.19. Leaf Superpixel struct.

The SLIC algorithm [21] is used to compute the superpixels. For the implementation itself I have used OpenCV [19] where the algorithm is already implemented. For the input image that will be divided into the superpixels we use the visualisation of structural distance. The algorithm outputs an matrix where each pixel has a label to which superpixel it belongs. All that needs to be done is to transform this superpixel representation into the struct LeafSuperpixel mentioned above. The algorithm also outputs the number of superpixels or rather it takes it as an imput. Using this information we traverse the image similarly as in LeafRegion part and get vector of regions each of which represents the area of the superpixel. Since the algorithm can add pixels that are out of our image mask, we mask the undesired pixels out. Doing so can create a lot of empty regions so we iterate through the vector and remove them. Last part is to compute the additional data we simply iterate over all of the points in each region and compute the mean color and distance. Here could be a space for experimentation to compute these information differently. The code for the generate function is in figure 3.20 and the output visualisation is in figure 3.21.

3.3.5 Region Selection

All we need for region selection we already prepared and described in previous sections. All we need to do now is the selection of similar region for the target image. It goes without saying that we have two sets of regions one for the target image and a one for the example image. We simply iterate over all of the target regions and select one from the example regions. It should be mentioned that it's not necessary to have the same number of regions in both pictures and one region from example set can be used more than one time.

The selection of similar region goes as this. For each example region we will compute the similarity equation (1). We keep the N best similar region in a vector. The actual amount of best regions that we keep is 20% of the number of regions. From the the vector of the best regions we choose randomly one and the chosen one is now a region that we will take color from. To know which target region chose which example region a vector of Ids is saved, where the position of the vector represents the target region position and the value save is an index in the vector that holds example regions. We can see a result of region selection in figure 3.22.

```
INPUT: cv::Mat& img,
       cv::Mat& img_color,
       cv::Mat& img_mask,
       std::vector<std::vector<int>> StructuralDistance,
       int numSuperpixel)
{
    slic.GenerateSuperpixels(img, numSuperpixel);

    std::vector<Region> superpixelRegions
        = MaskRegions(slic.GetLabel(), img_mask, numSuperpixel);

    superpixelRegions = RemoveEmptyRegions(superpixelRegions);

    std::vector<LeafSuperpixel> leafSuperpixels
        = ComputeStatistics(superpixelRegions, img_color,
          StructuralDistance);

    return leafSuperpixels;
}
```

Figure 3.20. Leaf Superpixel Generate function.

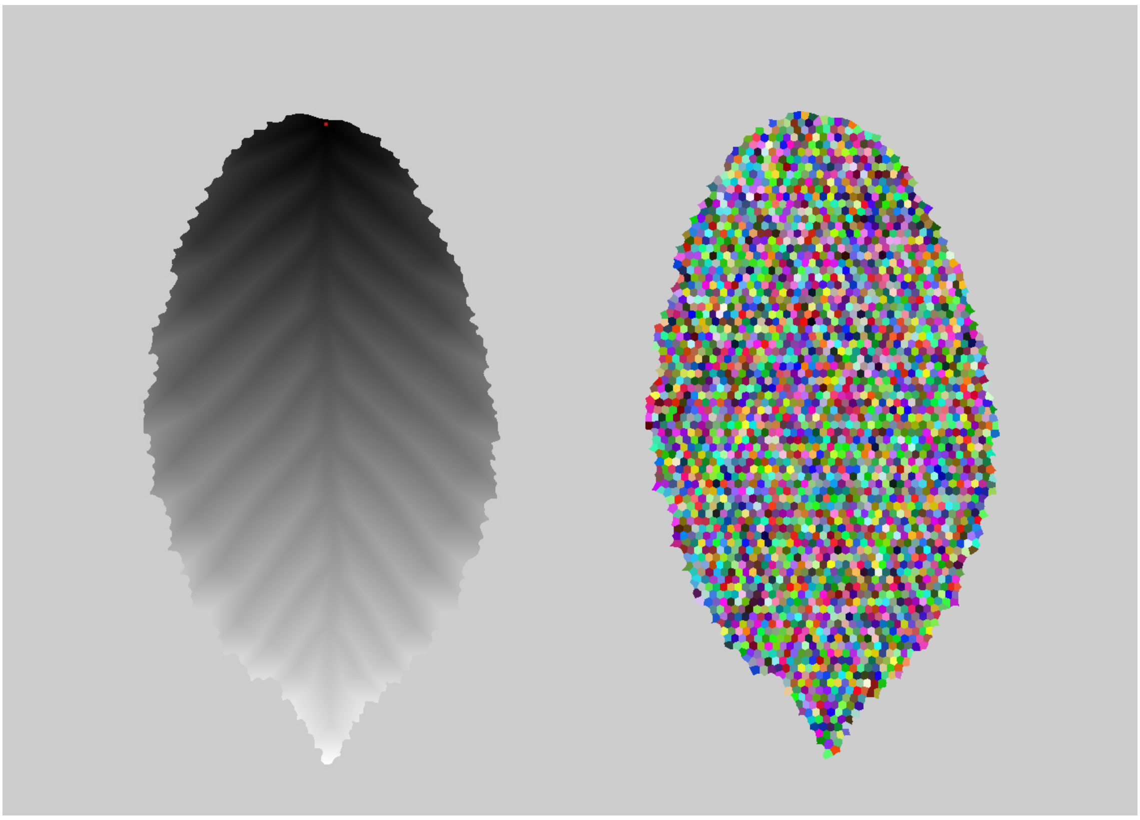


Figure 3.21. Superpixel result. Structural distance visualisation input on the left and superpixel output on the right.

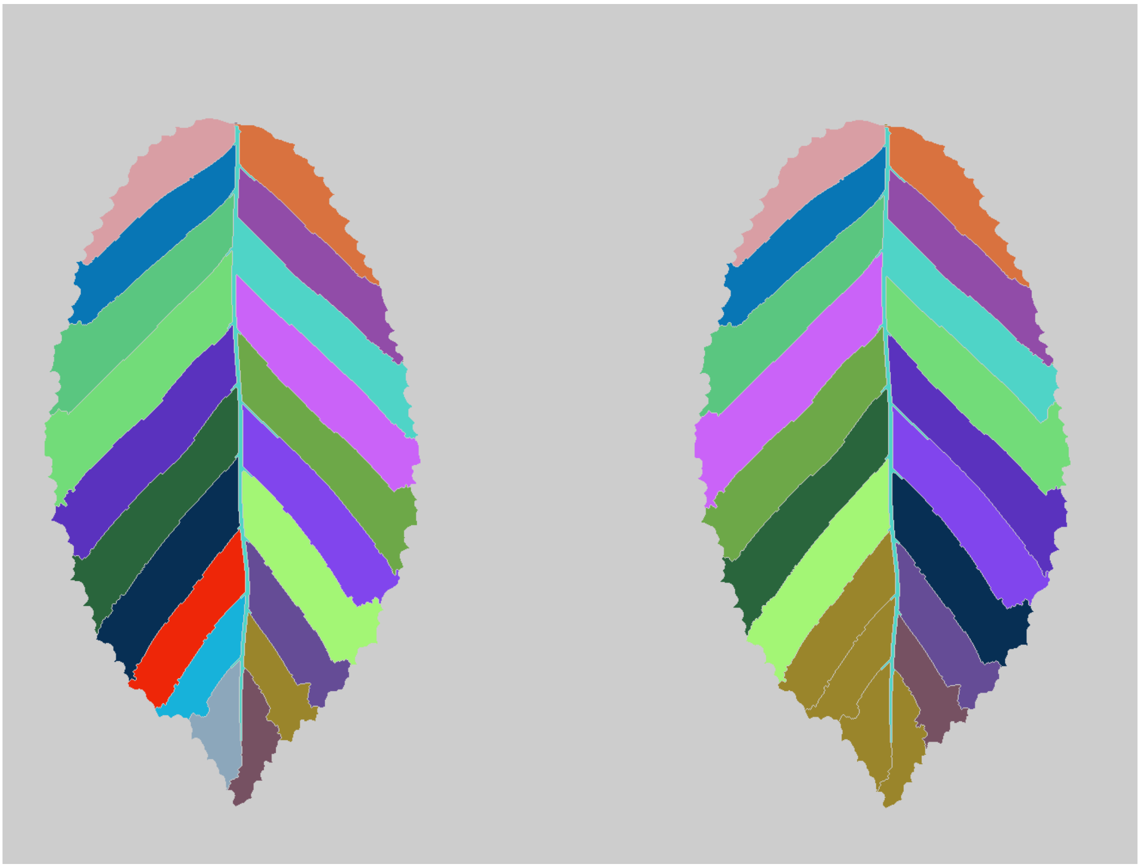


Figure 3.22. Region Selection. Example regions on the left. Selected target regions on the right.

3.3.6 Pyramid Creation

Now that we have the target regions matched with example regions we can move on superpixel selection. But before we do that we need to prepare superpixels in a way that is efficient and will provide good results. A pyramid style texturing is applied in superpixel creation similar to [22]. We create a pyramid that has in each level a higher quality of the image. The quality here is controlled here by the number of superpixels. We generate a five level pyramid texture, the five here is used based what work for authors of paper [3] but here is a space for experimentation. In the pyramid we save a color of the super pixel. Five levels of pyramid superpixels can be seen in figure 3.23. Pyramid also saves the structural distance matrix and labels matrix. The ***PyramidLevel*** structure can be seen in figure 3.24.

```
struct PyramidLevel{
  cv::Mat color;
  cv::Mat dist;
  cv::Mat labels;
  std::vector<LeafSuperpixel> superpixels;
};
```

Figure 3.24. Pyramid Level structure.

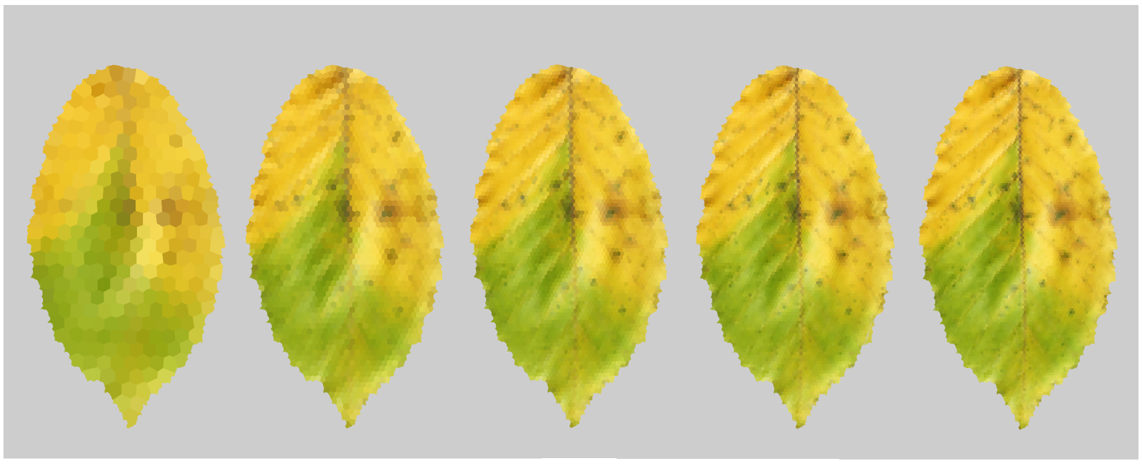


Figure 3.23. Pyramid Levels. From 1 to 5 with increasing detail (5 is the best).

3.3.7 Superpixel Selection

For superpixel selection we will use two pyramids one for the example and one for the target. The selection is done in level with increasing detail. In an iteration we find the superpixels that belongs to target and example region. Having now the superpixels we can start the selection. For each superpixel in a target region we take N most similar superpixels based on structural distance. Then we take M superpixels form the selection randomly. From these superpixel we take the best one based on how close is it to previous color. The first iteration selection is different then the rest of iterations since we don't have a color to refer to so the color matching is skipped. You can see the progression of superpixel selection 3.25.

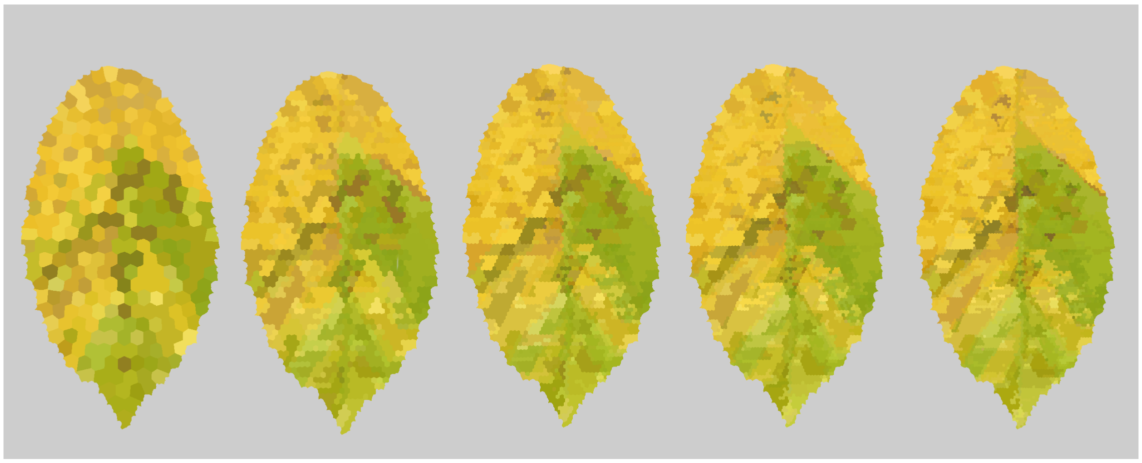


Figure 3.25. Selection progression. From 1 to 5 with increasing detail (5 is the best).

3.3.8 Image smoothing

After superpixel selection the image has a very hard transition between superpixels. To get a better distribution without the sharp edges we post-process the leaf texture image as described in [3]. We minimize the following energy function:

$$E = w_0 \sum_{i \in I} \| x_i - x_i' \|_2 + w_1 \sum_{(i,j) \in N} \| x_i - x_j \|_2 \qquad (2)$$

where I is the set of target pixels, N is the neighboring pixel pair set which in our case is the four pixels from the left, right, up and down directions. The first sum keeps the image close to the superpixel color and the second sum smooths the edges. We can tweak the values w_0 and w_1 to put emphasis on the two sum meanings. Since they are both competing against each other we can fix w_0 to one and tweak only w_1.

For the minimization of the function we use the Gauss-Seidel iteration. Where we based on the leaf image mask decide what pixels are relevant and look up the neighbouring pixels. The neighbouring pixels are not always four (when we are at the edge of the image for example) so we change the Gauss-Seidel kernel accordingly (we simply add only what is there and divide by one less). The equation that form the keren is the following:

$$I'[x,y] = \frac{1}{1+4w_1}(w_1(I[x+1,y] + I[x-1,y] + I[x,y+1] + I[x,y-1]) + b[x,y])(3)$$

where I' is the new image, I is the current image, x represent the row, y represents the column and b represents the superpixel color. We got the following equation as stated bellow:

$$E_0 = \parallel x_0 - x_0' \parallel_2 + w_1 \sum_{i=1}^{4} \parallel x_0 - x_i \parallel_2 \tag{4}$$

we need to know the value for single pixel, so to get the idea of how it works we substitute into the equation

$$E_0' = 2(x_0 - x_0') + w_1 \sum_{i=1}^{4} 2(x_0 - x_i) = 0 \tag{5}$$

to minimize the energy function we derive it and set the equation equal to zero.

$$x_0 - x_0' + 4w_1 x_0 - w_1(x_1 - x_2 - x_3 - x_4) = 0 \tag{6}$$

when we write the equation in the basic terms and put x_0 on the left side we can see that it corresponds to equation 2. Using this equation we can iterate in the Gauss-Seidel method until we convert to some minimum. In my implementation 100 was enough but this can be tweak to ones desire, since when it converts the image no longer changes. The result of the smoothing on the whole picture can be seen in figure 3.26 and the detail can be seen in figure 3.27.

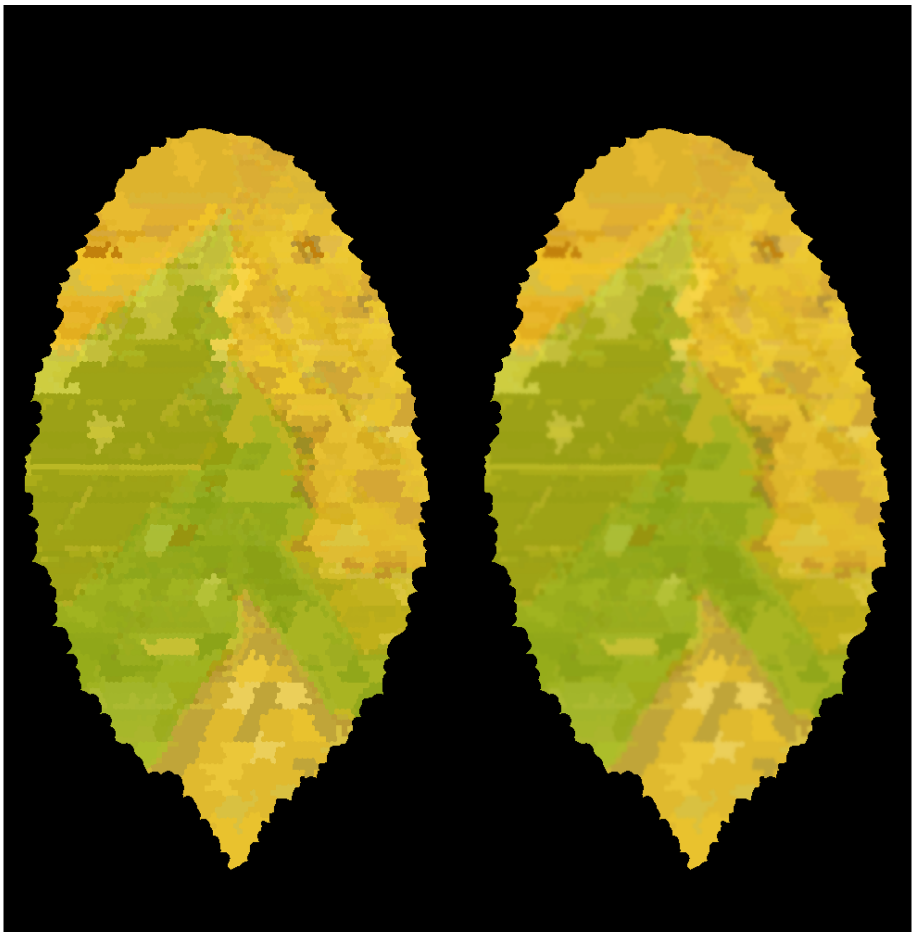


Figure 3.26. Smoothing. Left before, right after.

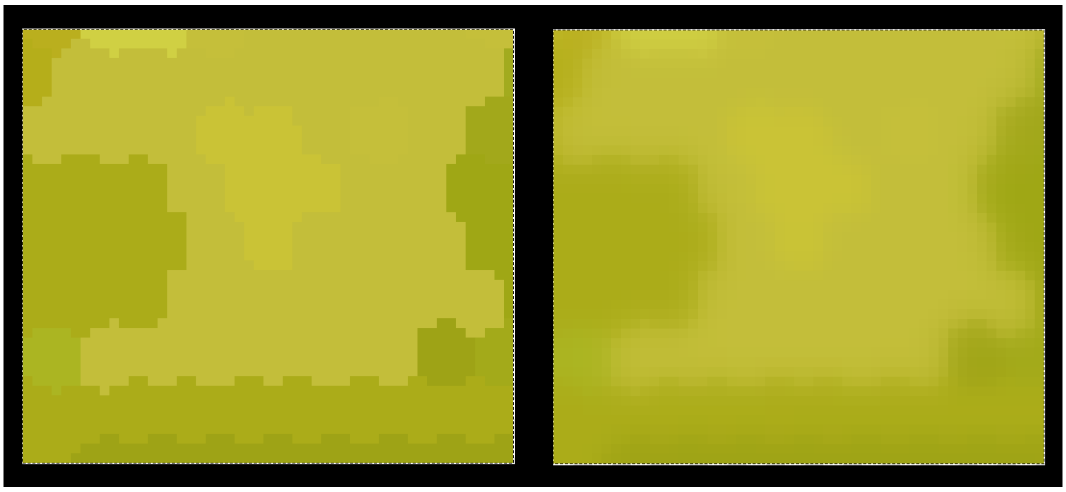


Figure 3.27. Smoothing detail. Left before, right after.

Chapter 4
Tree Simulation

In this chapter I will discuss a way how to simulate tree in the wind. There are two possible ways presented. One that was the original idea that I was not able to implement but my findings are presented there. The second way is presented and also described how I implemented it. For the simulation a special data format is followed. The modeling of such format is described in section Tree Data. To get a better simulation of the wind a simple way how to generate a wind field is presented. In the simulation we want to move the geometry of the tree according to wind to achieve that we use skinning algorithm and that is described in section Skinning.

4.1 Wind Field Generation

There are a lot of ways how to generate a wind field. One can use sophisticated algorithms to simulate the wind field that reacts to various attributes. Or one can use the data driven approach to collect wind data and interpolate between them. There is a lot of way of going about to create a wind field. I have chosen the simplest solution that came to mind that would create a pleasing results. I chose to use the 3D Perlin noise [23].

I use the continuity of the Perlin noise. In the z dimension I iterate through the 2D matrix values and since they are continuous the change will not be abrupt. To simulate the typical movement of the tree back and forth I simply go up to an index and then go back.

For Perlin Noise implementation I use open source noise generation library called FastNoise Lite [24]. I simply call a library method generate the 3D matrix and then index the given z dimension depending on the current time of the simulation. The output is visualised in figure 4.1.

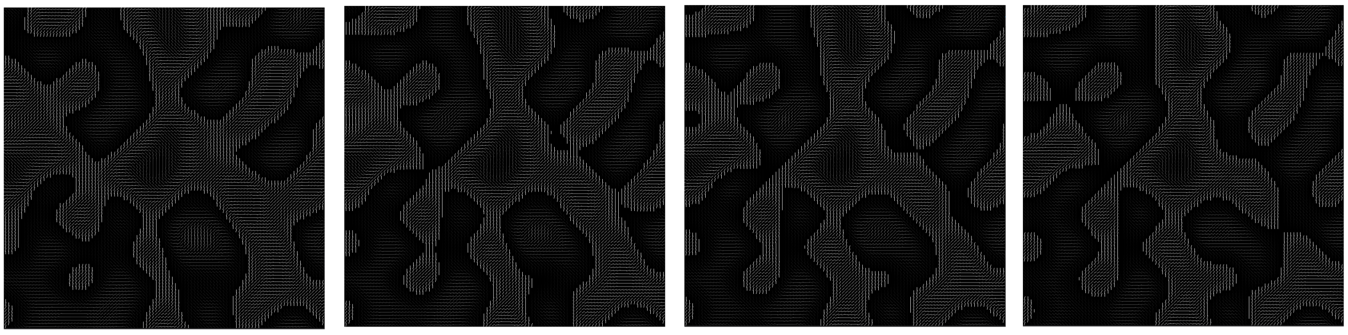


Figure 4.1. Wind Simulation visualisation. Evolving from left to right.

4.2 Skinning

One of the main focuses of my thesis is to make the tree move in the scene. To make things move we need to transform them in the scene in each frame to the desired position. We have a tree model that consist of vertices loaded using Assimp [25], but how do we get the transformations to get the vertices moving. This is where skinning or skeletal animation comes to play. What skinning basically does is define what transformation should be used on a given vertex. In this section I will try to explain how skinning does it. It is based by an excellent tutorial on learnopengl website [26].

In skinning we have two main components (or structures if you will) ***Skin*** and ***Bones***. Skin is simply the collective of meshes of the model. Nice way of looking at it its the visuals (what is displayed). Bones are a what makes the object move, similar to the human body bones (hence the same name). When a muscle moves a bone all that is connected to the bone will move with it. So bone has a part of body that it effects. Same with the model, but in the model case it is a set of vertices in the mesh that reacts. Another similarity is that the muscles make the movement but in the skinning case its the transform, a matrix that by multiply the vertex with it will move it in the space. How the model is obtained is discussed in section ??, but in short its modeled in Blender [27].

But how much effect does a bone on a vertex have and what happens when two bones effect the same vertex? Here is where the weights come in each vertex is effected by a bone with given weight that adds up to one (typically). So when the transformation of the bone is applied it is than multiplied by the weight to reduce the effect. This is not really intuitive and one could mistake it with the overall connectivity of bones. In body terms: when I move my shoulder the whole arm moves. This is not what bone weights are, that would be the parent-child relationship of the bones that I will touch on later in this section. The weight how strong of a connection between the bone and the given vertex is. To assign the weights for our model we use the Blender functionality to generate automatic weights, on can do this manually through weight painting mode but the automatic generation gives good enough results. In the figure 4.2 you can see a visualisation bone weights.

Bones are structured into a hierarchy where we keep track of the parent-child relationship. In general bones can be completely separate and not depend on each other at all, but in practice this rarely happens. Also since in this thesis we are simulating a real life tree we will also follow similar structure. Because of that we will limit our structure to a directed acyclic graph (DAG for short). This DAG will have one root from which all the bones start and each bone will have one parent (except root). With these restriction we could be more specific and say that the structure is a tree structure, but since the world tree would be present too many times in the thesis I will stick to DAG (Tree is a DAG, so we can do this). We have this bone structure, but how would you create a shoulder only movement that I described in the previous paragraph? This is where the parent-child relationship come in. Each child will first apply the transformation of the parent and than its own. This way we accumulate a transform along the way up to a given bone. In code term we recursively traverse the DAG and send our matrix transformation to the child, the child multiplies its matrix transformation by the parent one and uses this new one it also sends this new matrix transformation to its children. A simple bone structure that was used for testing is in figure 4.3.

Now that we know the basics of skinning we can describe the overall algorithm of skinning that is used in this paper. Normally skinning is used with animations and we interpolate between keyframes, but since this is an simulation we apply the simulated transformations in the frame where we computed them. So our rigid body simulation will give us a series of new transformation matrices for each bone. We save these

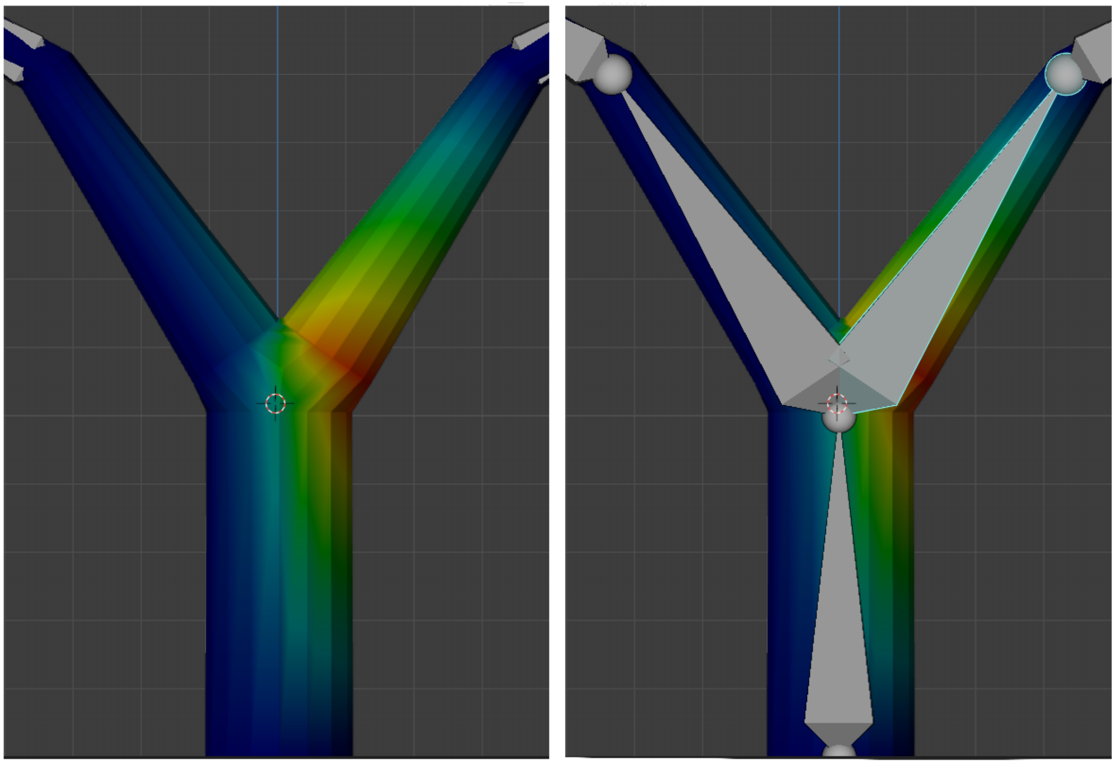


Figure 4.2. Bone Weight Demonstration. The effect of a bone visualisation, where red means 1 (copy exact transformation) and blue means 0 (transformation has no effect). The bone is highlighted with light blue outline in the right picture.

transformations matrices into an array. Since each vertex has information which bones with what weight effects it, we simply load all of these values (given matrix and weight) in vertex shader and apply the transform and weight and we have the vertex in new position. Then we proceed to transform the vertex normally as you would in the graphical pipeline (model, view and projection). A vertex shader code can be seen in figure 4.4

4.3 Tree Data

For rigid body simulation we need to define several attributes for our rigid body simulation (mass, length, center of mass, etc.). There are again several approaches how to do this a lot of papers generate their own geometry for the trees and hence generate these attributes during generation phase [28–29] because they have more control during this process. Second option is to reconstruct tree from by scanning it using a drone as in [30]. Here we also have more control when processing all the data gathered by the drone and we can deduct these data from this. Last approach is to model the tree in some modeling software and this is the path I chose. An example of how would a simple tree like this look like is in figure 4.5.

In my approach I didn't use L systems because they would add additional load into my work and also their structure is not as desirable. The scanning approach would prove to be even more difficult and out of the scope of this thesis. I chose a simple approach although a bit tedious, I will model the tree in blender [27]. I have created a simple atlas of primitives (trunks, branches, ...) that I will combine together to make

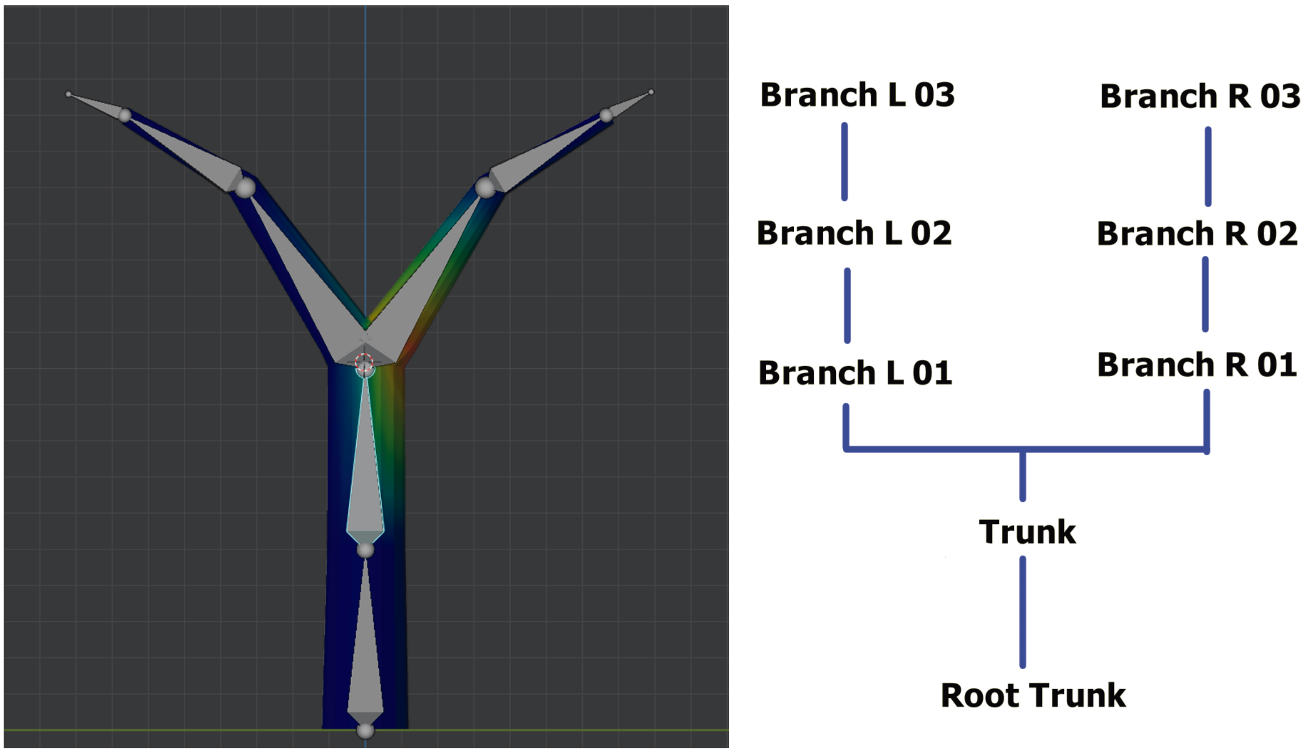


Figure 4.3. Bone Structure Example. Tree model for reference on the left. Structure visualisation on the right.

```
uniform mat4 finalBonesMatrices[MAX_BONES];
void main()
{
    vec4 totalPosition = vec4(0.0f);
    for(int i = 0 ; i < MAX_BONE_INFLUENCE ; i++)
    {
        if(boneIds[i] == -1)
            continue;
        if(boneIds[i] >=MAX_BONES)
        {
            totalPosition = vec4(pos, 1.0f);
            break;
        }
        vec4 localPosition = finalBonesMatrices[boneIds[i]]
                                * vec4(pos, 1.0f);
        totalPosition += localPosition * weights[i];
    }
    gl_Position =  projection * view * model * totalPosition;
}
```

Figure 4.4. Skinning Vertex Shader. Shader inspired from [26].

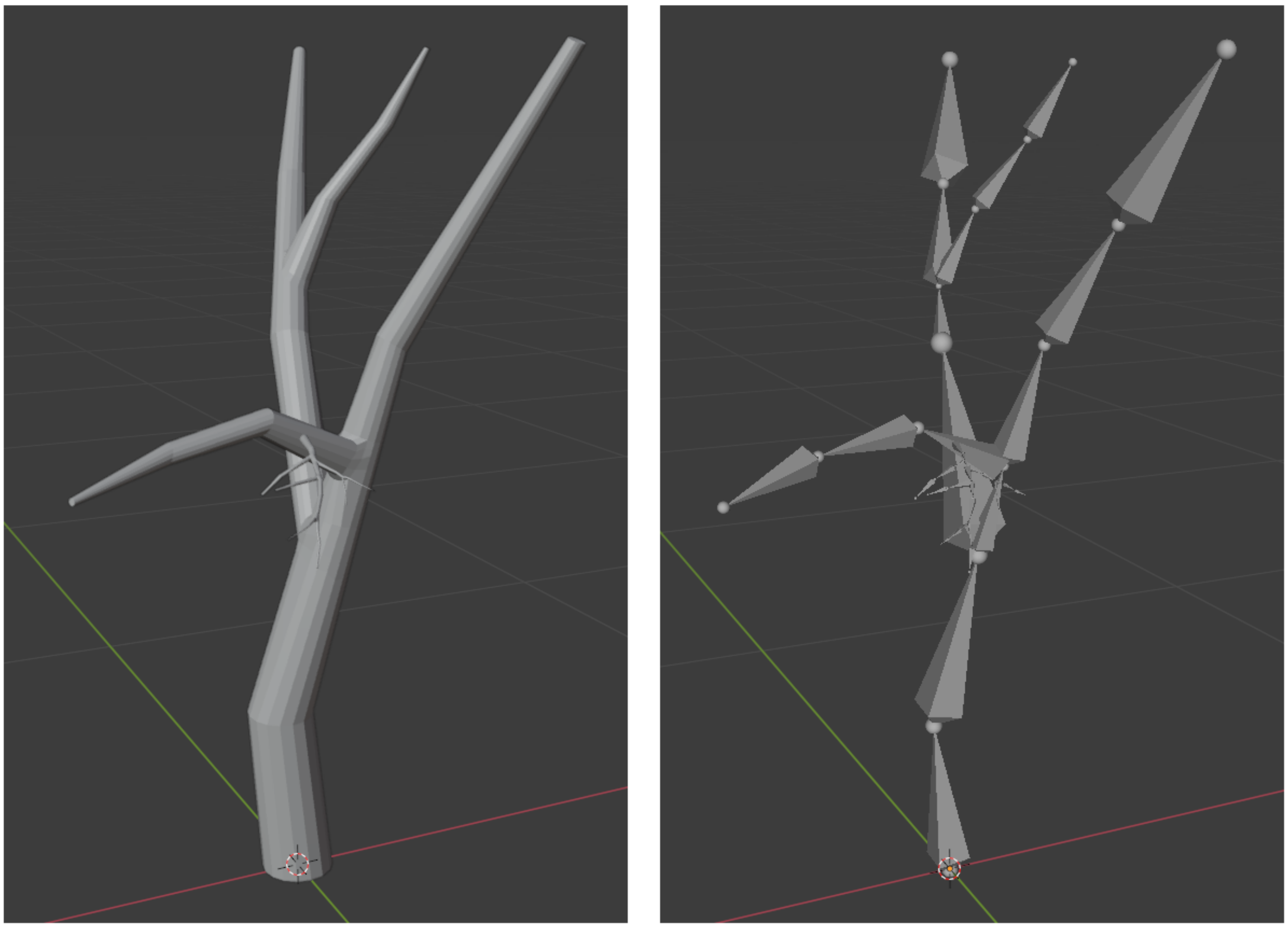


Figure 4.5. Demonstration of the blender tree model. Mesh for the model on the left. Bone structure of the model on the right.

the tree. You can see a sample of the primitives in figure 4.6 The geometry is pretty straight forward and there are even other software that have tree models available (as mentioned in [11]). But I also need to save information about structure of the tree and the rigid body primitives that discretize it. For that I use an armature a bone structure that can be exported in .fbx format. With that I will have necessary data.

To load the .fbx I use the Open Asset Import Library Assimp [25]. I save the data into class called ***Tree***. Tree has a model that saves the rendering related data (vertices, indices, VAO, ...) the code is based on popular tutorial [26]. We also use Assimp to load just the armature data. We locate the root bone of the that works as our articulate rigid body structure, you can imagine it as a directed acyclic graph.

Having the structure and model data we can now compute the rigid body attributes. These attribute are related to the algorithm that we are gonna use for rigid body simulation computation. We will use algorithm from [29] so all of the attributes are related to this paper. We will traverse the directed acyclic graph from root node to all of its children recursively. But first we need some additional information for the tree, we will need a bone map (that will tell us which bone is related to given rigid body), a vector of vertices (that will be used for calculation) and we also need to find the root bone from Assimp (a simple traversal of Assimp load structure). Having all that we can start computing.

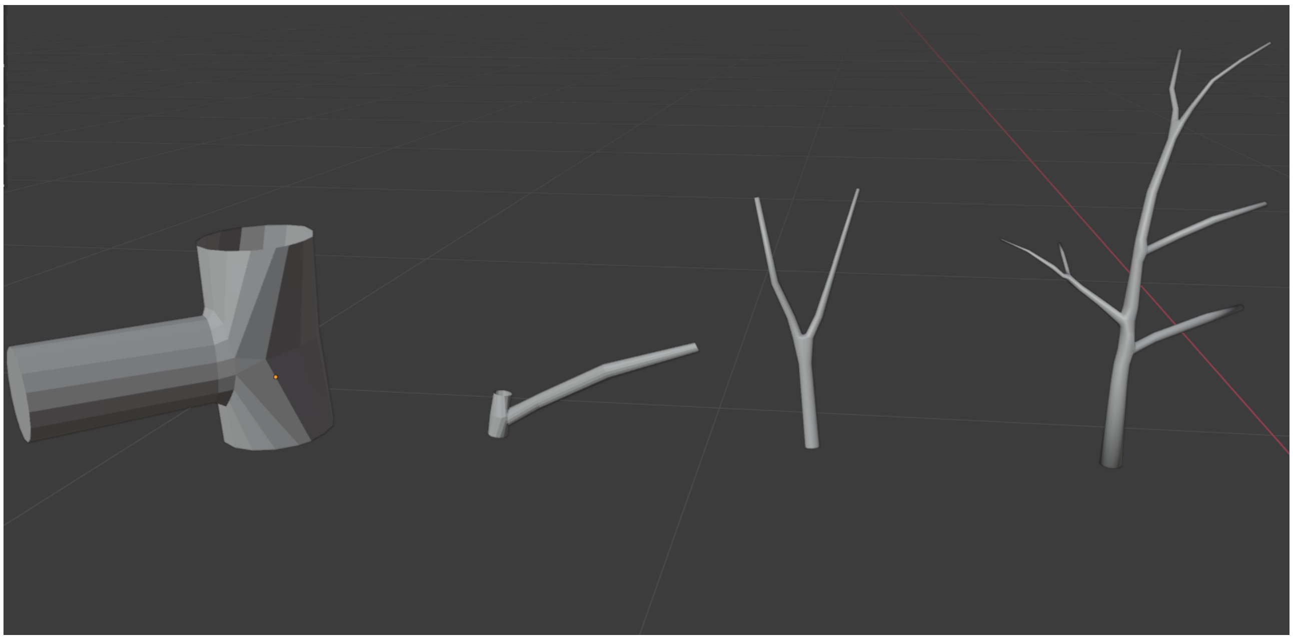


Figure 4.6. Blender atlas of primitives used to speed up the tree modeling process.

There are attributes that stay constant throughout our rigid body simulation and attributes that change every timestep. Because of that we divide them into two arrays, the constant one can be read only and have better memory properties. The two attribute structures can be seen in figures 4.7 and 4.8.

```
struct RB_Constant {
  int boneID; //ID of a bone that will be used for skinning
  float m; //mass
  float l; //length
  float S_f; //surface area
  float k; //rigidity of a branch
  float micro; //vibration suppression constant
  float Th; //thickness of branch segment
  int children_idx; //first child index
  int N_Children; //number of children, used to get all of them since
                  //they will be ordered one after the other
  glm::vec3 COM; //Center of mass relative to branch start
};
```

Figure 4.7. Rigid Body Constant Data.

First we compute ***Branch end*** *be* and ***Branch start*** *bs* because base on them we compute the radius of the cylinder which is our shape of option to represent rigid bodies. It matches well with trees because the are somewhat of a set for cylinders that are connected to each other. You can see demonstration in 4.9. Since the rigid bodies are articulated one *be* is another *bs* we use this to our advantage and compute the position only once. We compute *bs* matrix transformation of the origin (0,0,0,1) to given joint space. We know all the transformation thanks to the directed acyclic graph that we

```
struct RB_Constant {
  glm::vec3 v; //velocity
  glm::vec3 theta; //current orientation
  glm::vec3 omega; //angular velocity
  glm::vec3 K; //previous restoration force
  glm::vec3 bs; //branch start
  glm::vec3 be; //branch end
};
```

Figure 4.8. Rigid Body Changing Data.

read from the fbx tree file. So we traverse this DAG to all of its children recursively passing the accumulated transformation to the next child.

Using this we will get a global position of each rigid body joint origin and hence get the *bs*. There is only one problem and that is that to get the *be* of the leaf child we need one more bone, since the Assimp cannot read this information from bones alone (only transofrormation is saved). Because of that we need to add one more bone to the tree structure that does not have any effect in the skinning algorithm. With that we can easily compute the *bs* of the last bone and use it as *be* from the previous. In post process we then remove these bones (the ones that have only one child).

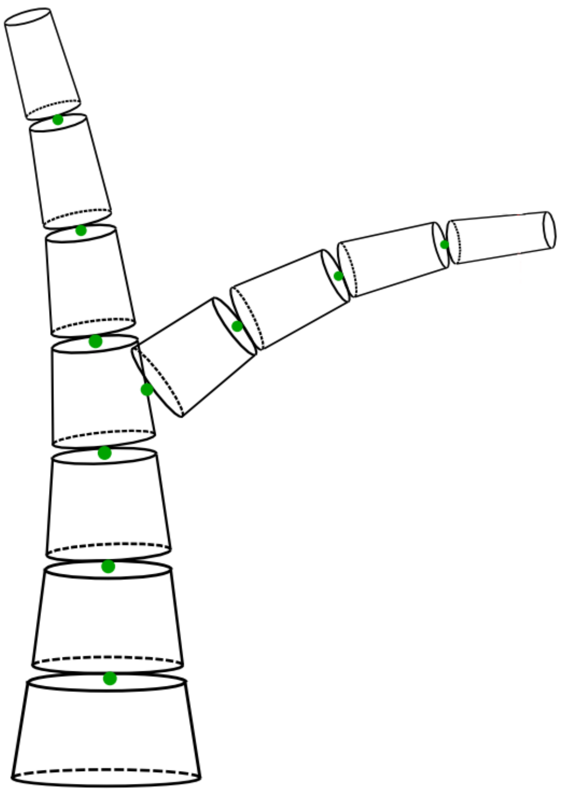


Figure 4.9. Tree Cylinder Structure demonstration. Green points represents the joints that connect the rigid bodies. Our *be* and *bs*. The original image is taken from [11].

With the bs and be computed we can compute the ***Length*** l. We compute it simply by subtracting the be from bs and computing the length of the vector. Another thing that we can compute with the help of bs is the ***Radius*** r of the cylinder. We will also need the vertices that are related to the cylinder (that the bone effects). With these two things we compute the distance from bs to all of the vertices. Sort the distances and take few closest vertices and compute the average distance. This average distance is our radius.

The logic here is that since our bs will be approximately in the middle of the cylinder and the closest point will be the most parallel to it and hence will be our radius. We take multiple points and compute average because the cylinder base is a circle and the closest points will have similar distances. Thanks to this outliers will be eliminated. The vertices are in global space since bs is also, so we only apply the model transformation to them so that they are correct.

Center of mass COM is computed by as an average position of all of the vertices that are related to the the given rigid body. We treat each vertex as an particle with same mass (here 1) and compute the COM in normal fashion as the average of all of the particles.

Having r computed we can compute some other attributes. ***Mass*** m is computed by computing the volume of a cylinder $V_{cylinder} = \pi r^2 l$ multiplied by tree wood density constant. To get the idea what kind of number we should use website Matmatch [31] but further tweaking with constant is necessary to get the desired result. ***Surface area*** S_f is also computed from cylinder equations for surface, where we only take into consideration the curved surface area $S_f = 2\pi rl$.

There are three more attributes that are computed from r and according to [29] they are Rigidity of a branch, Vibration suppresion constant and Thickness of branch segment. ***Rigidity of a branch*** k in the mentioned paper [29] they tested several values k and ended up with equation $k \approx r^{2.5}$. ***Vibration suppresion constant*** μ is similar to k and as mentioned in the paper [29] is computed as $\mu \approx r^{3.5}$. The last one related to r is ***Thickness of branch segment*** Th and that is computed simply as a diameter of the cylinder $Th = 2r$.

We also need to save the ***Bone ID*** $boneID$ so that we can then create the array of transformation for skinning algorithm. Thanks to Assimp that loads the ID for us we can simply get it from the assimp Bone node structure. The next attributes are changing every simulation step but initially are all set to zero. So ***Velocity*** v, ***Current Orientation*** θ, ***Angular Velocity*** ω and ***Previous Restoration Force*** K are all set to zero vectors.

Now we have all the initial attributes computed for our algorithm. In the end we change the tree structure to an array so that we can work with it better at GPU. The tree flattening into an array is done in an BFS manner so that the related children that will be loaded are close to each other in memory. An image presenting the principle can be seen in 4.10. In the image you can see that we have saved only index of the first child and thanks to the data cohesion we can deduct the children simply by iterating along the array until we reach the last child. To know when to stop we save an integer of the amount of children.The overview of the initialisation process can be seen in figure 4.11

4.4 Tree Simulation

Let me begin to say that implementation of the simulation proved very difficult and it is currently not working 100%. I also had to change my whole approach because math

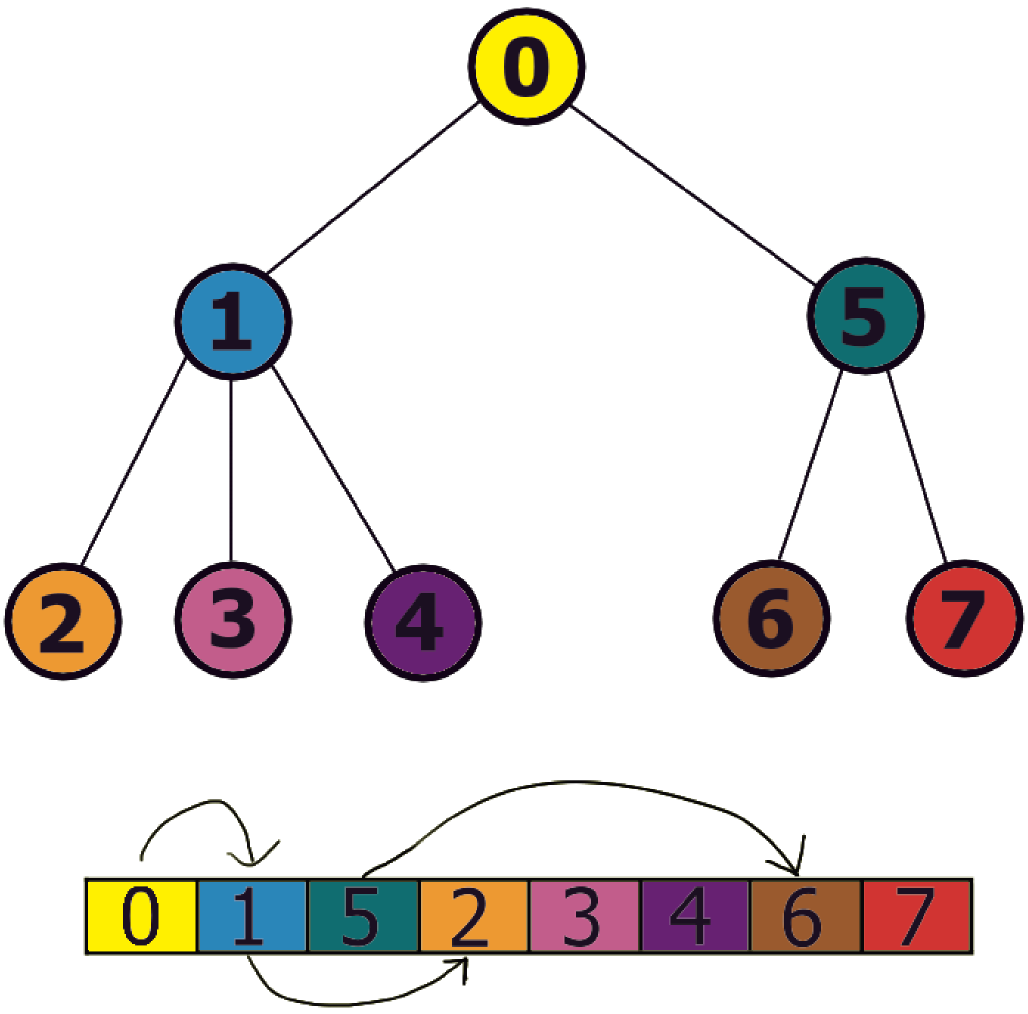


Figure 4.10. Tree Flatten Example.

proved too difficult for me to implement. There are issues with this part of my thesis but in this section I will provide all of my findings and what I think went wrong.

4.5 Original idea for implementation

Originally my plan was to adapt the algorithm from Real-time Interactive Tree Animation [11]. The paper claims to have very attractive results mainly that its fast and the resulting trees look surprisingly natural. This all revolves around its method of computing the attributes for equations of motion (angular velocity, angular acceleration, etc.). It uses an articulated rigid body structure to represent the trees and proposes an O(N) algorithm. The idea is to alleviate any time steps restrictions by using an analytic solution. The method follows 4 steps:

External force computation First we need to all the external forces and torques that are applied to the rigid body. Forces are simple, all one needs to do is to compute the sum of all of the affecting forces. $f_p = \sum_i f_{p,i}$. $f_{p,i}$ is the force of the rigid body p and i identifies the index of the given affecting force. Torques are a bit more challenging but not that hard either. Again it boils down to a sum $\tau_p = \sum_i r_{p,i} \times f_{p,i}$. Here the vector

```
void LoadTree(std::string path)
{
  //Read the Model rendering data
  model = Model(path);
  //Read the bone structure data
  const aiScene* scene = assimpImporter.ReadFile(path,
                                                 aiProcess_Triangulate);
  //Find the root bone
  auto rootBone = FindBoneRoot(scene->mRootNode);
  //get skinning bone info
  std::map<std::string, BoneInfo> boneMap = model.GetBoneInfoMap();
  //get all vertices
  std::vector<Vertex> vertices = model.GetAllVerticiesArray();
  //first pass to compute bs and be, relevant vertices for each bone
  RigidBodyData rootRigid = CreateRigidBodyData(rootBone,
                                                boneMap,
                                                vertices);
  //compute all relavant attributes and save the tree structure into
  //an array
  auto ret = GenerateRigidBodyDataList(rootRigid);
}
```

Figure 4.11. Tree rigid body initialization algorithm overview.

$r_{p,i}$ is vector from the articulation point to where the force is applied. Here we use the somewhat unspoken way of computing torque by vector cross product. This vector will tell us in radians how will the point rotate. This is easily parallelizable since both of these operation are not dependent on other rigid bodies.

Composite body update Since our tree is composed of articulated rigid bodies we need to compute the effects of the connected children on all rigid bodies. So we need to compute the effects that the children have on a rigid body. Essentially it is a tree traversal from leafs to root where you compute children contributions by computing these equations:

$$\hat{m}_p = m_p + \sum_{c \in C_p} \hat{m}_c \tag{1}$$

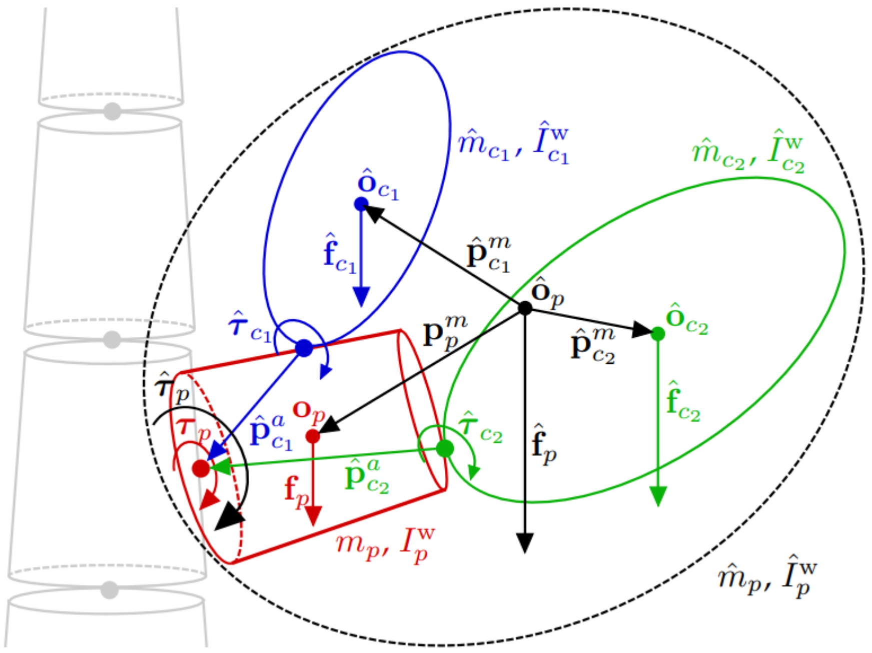
$$\hat{I}_p^W = I_p^W - \hat{m}_p \hat{p}_p^{m*} \hat{p}_p^{m*} + \sum_{c \in C_p} \hat{I}_c^W - \hat{m}_c \hat{p}_c^{m*}{}_c^{m*} \tag{2}$$

$$\hat{f}_p = f_p + \sum_{c \in C_p} \hat{f}_c \tag{3}$$

$$\hat{\tau}_p = \tau_p + \sum_{c \in C_p} -\hat{p}_c^{a*} \hat{f}_c + \hat{\tau}_c \tag{4}$$

All of these equations have in common that they sum their children contributions. There are few tricks used here, that are not explained in the original paper whatsoever and these are the computation of moment of inertia matrix by $\hat{m}_p \hat{p}_p^{m*} \hat{p}_p^{m*}$. To understand this you need to first know that the * means that its in skew symmetric matrix. If you put two multiple the vector $\hat{p}_p^m$ with in the skew matrix you will get the inertial matrix for cylinder. There is a second trick in the equation 4 and its again about the skew matrix here its used as an way to compute cross product. To get a visual idea how things work you can take a look at figure 4.12.

This part of the algorithm is not good for parallelization, but some can still be achieved one could hold a list of all the children for each level and start as many thread as there are items. This way you can achieve per level paralelization.



$\hat{m}_{c_1}, \hat{I}^{\mathrm{w}}_{c_1}$
$\hat{m}_{c_2}, \hat{I}^{\mathrm{w}}_{c_2}$
$\hat{\mathbf{o}}_{c_1}$
$\hat{\mathbf{f}}_{c_1}$
$\hat{\mathbf{p}}^m_{c_1}$
$\hat{\mathbf{o}}_p$
$\hat{\mathbf{o}}_{c_2}$
$\hat{\boldsymbol{\tau}}_{c_1}$
$\mathbf{p}^m_p$
$\hat{\mathbf{p}}^m_{c_2}$
$\hat{\mathbf{f}}_{c_2}$
$\hat{\boldsymbol{\tau}}_p$
$\boldsymbol{\tau}_p$
$\hat{\mathbf{p}}^a_{c_1}$
$\mathbf{o}_p$
$\hat{\boldsymbol{\tau}}_{c_2}$
$\hat{\mathbf{f}}_p$
$\mathbf{f}_p$
$\hat{\mathbf{p}}^a_{c_2}$
m_p, I^{w}_p
$\hat{m}_p, \hat{I}^{\mathrm{w}}_p$

Figure 4.12. The red rigid body has two children, whose composite rigid bodies are shown as blue and green ellipses. The center of mass of each composite rigid body is denoted by a point oˆ. The black, dashed ellipse denotes the composite rigid body corresponding to the red rigid body. Other labeled values correspond to the quantities in Equations 1–4. Image taken from [11].

Analytic spring evolution Now we have all the necessary data for rigid body computation. First everything is moved into the joint space and is hence noted with tilde (˜). We move to the join space by multiplying by the rotation matrix R which was computed in previous frame and will take us to the joints space. We also define new matrix K that represent the stiffness of the rigid joints. Having all that we get the analytic equation of motion:

$$\tilde{I}\ddot{\theta} + (\alpha\tilde{I} + \beta K)\dot{\theta} + K\theta = \tilde{\tau} \tag{5}$$

And here is where the confusion happens for me. In the paper they describe that the α and β are user defined parameters. But then they say that `$\alpha$ is set identically to 0` since they don't consider `ether drag` (which is probably aether drag). Even more confusion arrises when the solution for the equation 5 is presented. The paper uses eigen vectors to solve a differential equation (we want to get θ). In doing so they introduce I believe another K matrix which is what I think since I saw this is a common practice when solving differential equation using eigen vectors. But math is not my great suit and here I have no idea how to procceed. The paper further provides a derived equation after applying the eigen decomposition and more even more equations are provided.

But like I said I got confused here and I don't know how to proceed further. I had to go for another approach which will be described in the following section. Never the less if I managed to implement this solution I could achieve $O(N/Threads)$ paralelization since there are not dependencies.

Rigid body state update is the last step of the we compute the equation of motion for rigid body. Given we know $\ddot{(\theta)}$, $\dot{(\theta)}$ and θ from the previous step we just substitute into the provided equations from the paper. Here we need to go from the root to the children because when a parent moves the connected children will move with it (you can imagine branch on a trunk). I will not write all of the equations here since they are in the cited paper [11]. Here the since there is dependency on the children again we can use similar approach as in the Composite update step and achieve some paralellism.

The paper then further goes into specific computation of forces, collisions etc. which is not really relevant further. Even though I failed to implement this paper it gave me an inside into the principles how thing will be implemented and sped up my adoption of the different second tree rigid body simulation paper ,[]RealtimeForestAnimationRigid. I did quite a bit of research and spend a lot of time trying to understand how things work so I didn't want it to go to waste so I wrote this chapter. Hopefully if someone tries to implement this paper this will help in any shape or form.

4.6 Tree simulation

To simulate reaction of a tree I followed the steps from [29]. For every rigid body I will compute the rotation vector θ. Each element in vector θ represent rotation value in the given axis ($\theta.x$ is the rotation around the x axis). This step is called the Dynamics calculation (we are computing the dynamics of the rigid bodies after all). Having the θ I will then go into the second step called Integration of Movements, where similarly to any articulated rigid body algorithm I will propagate from root to leaf the rotation. Well having all of these rotation is nice and all but how do I propagate them into the geometry? How will they effect the render? That is where skinning comes into play. Essentially what happens is from the θ I will compute a rotation matrix (a transform) and apply it to the bone that is connected to some geometry. With that the rotational changes will rendered. A high level algorithm flow is shown in figure 4.13.

4.6.1 Dynamics calculation

Now lets discuss the details of Dynamics Calculations. First of all its done in a CUDA kernel. I choose CUDA for the support of c++ modern features and because it is not that different from programming in c++. There is also a lot of resources regarding CUDA. The data is loaded from CPU data structure tree into the GPU using the $cudaMalloc$ and $cudaMemcpy$ functions. How data looks and how it is acquired is described in the chapter Tree Data. To put simply I have two arrays that hold rigid body data that changes and that is constant and a one array that describes each level

```
void TreeSimulation()
{
    tree = LoadTreeData();
    while(true){
        DynamicsCalculation(tree, deltat)
        boneMatrices = IntegrationOfMovements(tree);
        Render(boneMatrices, tree);
    }
}
```

Figure 4.13. Tree simulation overview.

of the DAG of the tree. Having all this loaded we can proceed to compute all the equations.

As stated in chapter Tree Data each branch is approximated as a cylinder. I will follow notation from the paper [29] to avoid confusion between renaming in code I also follow the same names. In the text bellow I will describe the given math equation and what relevant part of it can be implemented in the flow of the kernel that implements the Dynamics calculation First we need an equation of motion for the cylinder around an origin point O. It is as follows:

$$N = \frac{ml^2}{3}\frac{d\omega}{dt} \tag{6}$$

Here the N is the moment of force (also called torque), ω is the angular velocity, m is mass and l is length.

Here we again use the trick with the fact that torque can be computed using the vector cross product. So N can be expressed as $N = F \times c$, where F is the force applied onto the rigid body. We are also do a further approximation, where we use the center of mass (COM for short) as the one particle that all the rigid body computations will be on. Using COM has various good uses for rigid body computation, mainly that we can compute it as a particle and not compute specific collision points of the applied forces. c is the vector starting from the joint start (I called it bs in the tree data chapter, or it could be understood as the origin of the given rigid body bone space) to the COM. The new equation that we get is:

$$F \times c = \frac{ml^2}{3}\frac{d\omega}{dt} \tag{7}$$

Having the equation we can compute the parts of it, here the first relevant part is the c computation which is simple vector subtraction. We could also precompute $\frac{ml^2}{3}$ (since we know all of the elements) but as you probably guessed we will ultimately reorder this equation to compute $\frac{d\omega}{dt}$ which is the angular acceleration. From that we can compute backwards angular velocity and the angle that we should rotate the rigid body by. So precomputing $\frac{ml^2}{3}$ could be done this early but I chose to put in the end in a long equation.

Since we want to compute $\frac{d\omega}{dt}$ and we know c and $\frac{ml^2}{3}$ only F remains to be computed. As the name indacates F are the forces that are applied on the rigid body (specifically at the COM). So what are the forces that we take into account? They are in the following equation:

$$F = F_{wind} + K + R + T \tag{8}$$

The forces are the Wind force F_{wind}, Restoration force K, Axial damping force R and Back propagation force T. I will go through what each of them mean and how to compute them in the following text.

Wind Force F_{wind} represent the effect of the wind on the rigid body and its the main driving force that moves the object the rest of the forces react accordingly to the wind force effect. The force is sampled from a wind field as a velocity vector v. For the sake of speed we approximate the area effect of the wind by scaling it by the surface area of the branch. The force is applied on the COM particle as mentioned before. The equation for the wind force is:

$$F_{wind} = S_f \sigma v \tag{9}$$

Here the S_f is the surface area of the branch and σ (how to compute it refer to chapter Tree data) is the air viscosity coefficient constant (the value is taken from the internet). The implementation of this is straight forward we just follow the equation all the necessary values are known.

Restoration force K is a force that tries to restore the branch to its original position. It is proportional to the angular displacement from the original orientation. The equation is as follows:

$$K = k(\theta - \theta') \tag{10}$$

where k is the rigidity of the branch (again how to get k is in Tree Data chapter), θ is the original orientation (from the code perspective its a zero vector since at the start the tree is in the bind position without any rotation) and θ' is the current orientation. The equation itself is not dependent on its parent, but for computing the Back propagation force we need to know the current child Restoration force (when a child wants to go back to its original place the parent branch is also effected by the force). To keep Dynamics calculation independent for parallelization we take the value from the previous step. But this means that we need to compute this force in the second Integration of movement step. We can do this because the force has a small effect on the overall simulation.

Axial damping force R is force that binds the branches together. It is a force proportional to the square of the velocity. It damps the movement of the branches by gradually suppressing the vibrations of the branches caused by the external force. The external force is computed with:

$$R = -\mu\omega|\omega| \tag{11}$$

where μ is a constant determined by the thickness of the branch (how its computed is in Tree Data chapter) and ω is the angular acceleration. By itself axial damping force can be large and instead of damping the motion it can even reverse it. To ensure that the negative acceleration would not exceed ω we clamp it magnitude by:

$$|R| = min(\mu\omega^2, I\omega) \tag{12}$$

with this the force is not as significant. Implementing this is quite straight forward the clamping is done by first normalizing the vector R and then by multiplying it by $\mu\omega^2$ or $I\omega$ depending on what is smaller.

Back propagation force T is a force that propagates the restoration force to the parent from the children. The equation is as follows:

$$T_{i-1} = -\sum k_i K_i \tag{13}$$

T_{i-1} is the force of the given parent, K_i is the child's restoration force and k_i is the propagation coefficient that is determined by:

$$k_i = k_c \frac{Th_i}{Th_{i-1}} \tag{14}$$

where k_c is the fixed propagation coefficient and Th is the thickness of the branch. Back propagation force depends on the Restoration force from its children so to keep the code better paralellisable we take the K from the previous simulation step.

Now that we computed all the necessary data we can compute the ***Equation of angular motion***. We are working in classical coordinate system (x,y,z) as our basis. Which means that θ, ω and α all have three components. We know that $\alpha = \frac{d\omega}{dt}$ so we substitute it into equation 7 and separate the α on one side. We will get:

$$\alpha = (F \times c) * \frac{3}{ml^2} \tag{15}$$

Now that we have *alpha* we can go backwards and compute angular velocity ω as and the rotation vector *theta* as:

$$\omega' = \omega + \alpha(\triangle t) \tag{16}$$

$$\theta' = \theta + \omega(\triangle t) + \frac{1}{2}\alpha(\triangle t)^2 \tag{17}$$

Having computed these new values we save them into the changing rigid body data array. Now we have the rotation of the rigid body in θ and have it saved. But we don't rotate the rigid body here already since we need to propagate it and this is done in the second step Integration of movements. One might ask why can't we just rotate the rigid bodies here? Well because we need to propagate the effect of the parent to the children. An illustration of the rotation propagation can be seen in figure 4.14.

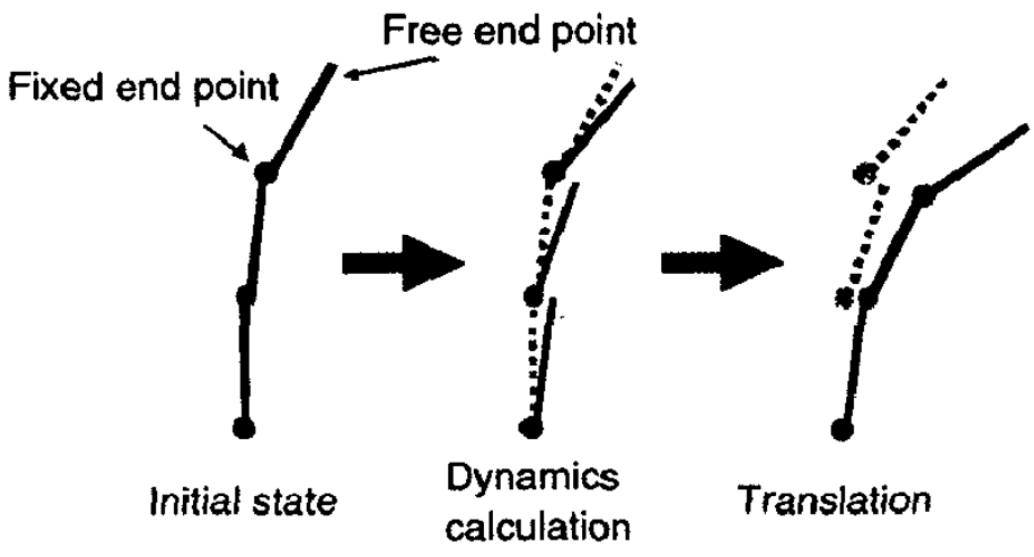


Figure 4.14. Illustration of how the rotation propagation works and why you should use it. Image taken from [32].

4.6.2 Integration of movements

We know now the how we should rotate each rigid body, all we need to do now is to systematically update the rotation. This is also done in a CUDA kernel but this one is started a bit differently. In the previous Dynamics calculation kernel we started as many threads as there were rigid bodies and thanks to the data being independent. But here the data is dependent so we need to approach it more systematically.

We will need a way to process the tree in levels luckily we prepared a structure for it already when we loaded the tree model. The vector of RBhold the necessary information. We are using the fact that the rigid body data is specially organized in a way that every child is in a same level is also next to each other in the array. Thanks to that we can just save the starting index of the level and number of children in the level. And this information is stored in RBall we need to do now is start the kernels. We do that in a for loop where we start as many kernels as there is rigid bodies in that level, we also send the start index of the children as parameter so that we can get the correct rigid body index from the kernel ID + the start index. We also need to wait for all of the threads in the level to finish so we call $cudaDeviceSynchronize()$ at the end of the for loop.

The Integration of movements kernel is simpler than the dynamics one. We just need to compute the restoration force for each rigid body. Since we will go through all the rigid bodies again we can do that and thanks to this we avoid the need to create some swaping logic between the previous K and the new one. If we computed it in the previous kernel it could lead to some weird behaviour where the K is read from some thread but it was already changed in different one and we would lose the previous value. Next thing that needs to be computed is the transform or rather the rotation of the rigid body, rotate relevant changing rigid body attributes and we also need to propagate the transform further.

The transform is computed from the θ vector. Each element of the vector represents how much we should rotate around the x,y,z axes. To compute the rotation matrix I have used the glm function $eulerAngleYXZ(rad_y, rad_x, rad_z)$.

Having the transform we can now transform the changing rigid body attributes. The attributes are the branch end be and the center of mass COM. We cannot just multiply the vector by the rotation matrix because the transform is computed relative to the branch start bs (branch origin). First we need to subtract the origin from lets say be (the other follow the same steps) after that we can rotate using the rotation matrix and last we need add the origin back so that its in the correct position in global coordinates. To be throughout COM is follows the same steps.

The propagation of the rotation is simple because we traverse the tree systematically (parents are always finished before children) we can just add the rotation to the children. So in each kernel we iterate through all of the children and add our θ to them. In this loop we also update the branch start bs of children we rotate it same as we rotated COM and be. Thanks to that all of the values are propagated and we get the correct new tree state.

The last thing that the computation does is compute the bone transform matrix that we use for skinning. This proved a bit difficult because working with different spaces is always a bit tricky. For the skinning algorithm we need to move into bone space and character space and for that we first need to use the bone offset matrix. Bone offset matrix will move as from the bind pose space into the global space. Now being in global space we can use the bone matrix to move to the bone space and then we can use the transform that we computed. In code its a simple multiplication of tree matrices:

$$skinningmatrix = transformmatrix * bonespacematrix * boneoffsetmatrix \tag{18}$$

but believe me it took me a bit of thinking and testing out before I figured it out. But now we have the resulting array of matrices which we can set to vertex uniform and compute the skinning like we normally would.

4.6.3 Results

In this section I have failed on more than one occasion I have to say that the simulation doesn't work. Or rather I believe it works but it is not stable. The idea is all there and it is connected well to the rendering pipeline through skinning. The rotations computed in the kernels are projected correctly onto the tree mesh. The issue is with the rotational matrix itself. The problem is that the wind force is too strong and the rotation doesn't stop and begins to rotate over and over and over. This lead to an improper state that you can see in figure [].

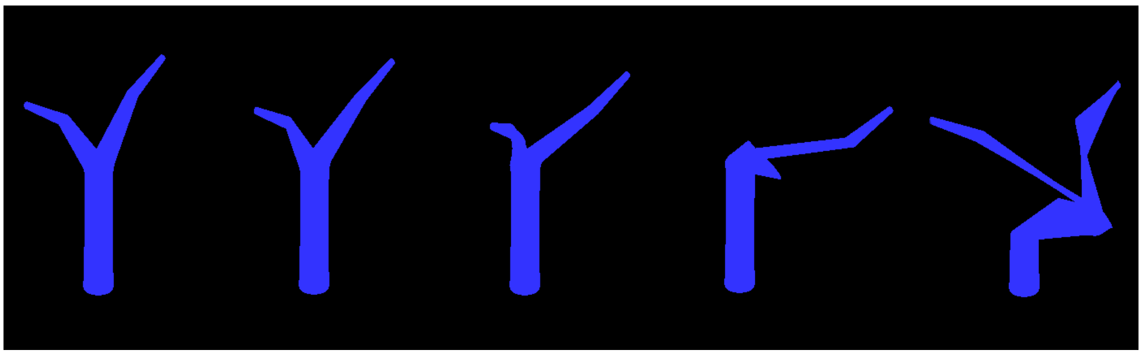


Figure 4.15. Tree simulation result. Time frames begins from left to right. You can see that the tree over rotates.

There could have been a lot of things that could have gone wrong so I am not entirely sure where is the issue. I believe I followed the paper [29] well and correctly. But in the paper there is not stated the actual values of the rigid bodies, so I am guessing the values that could be one of the reasons of why it doesn't work. I am basically guessing here what the values should be and I could spend a lot of time tweaking these values, but I simply don't have that time. Another thing is that the paper could simulate the values with really small time steps or they could have some magic numbers that made it work like it did or it could be some error in my code.

Nevertheless it doesn't work on 100%. One could hot fix it using the limits on how much the tree benches can bend (rotation limits) but it would not fix the underling issue with the simulation. This could be investigated further in another work but it would lead to a tedious work or it would need someone who understand math more and implement the solution from [11]. This is an issue with lack of a full picture and I simply don't have enough information or I am not smart enough to understand the math.

Chapter 5
Leaf Falling simulation

Falling leaves are a common feature often shown in games and movies and hence it has been studied for many years in many fields (not just computer graphics). However it is difficult to simulate accurately because it is quite a complex motion. In computer graphics it take it a step further where we want to have it fast in realtime. The most common way how to simulate a movement of a falling leave is to follow a path template that was created beforehand by an artist. But this way is unsuitable for GPU parallel processing and CPU lacks the computing power to simulate a lot of leaves.

In this chapter I adapt the work from GPU based real-time simulation of massive falling leaves [9]. The mentioned paper generates falling leaves paths automatically using combination of equations and collected data of real falling leaves. These paths are then divided into primitive motion sets which are then send to GPU and followed by the leaves. Each leaf will choose one of these motions to follow. To better understand the generation of leaves refer to Real-time simulation of lightweight rigid bodies [33] on which GPU based real-time simulation of massive falling leaves is based on. GPU based real-time simulation of massive falling leaves takes the original idea further by using velocity and angular velocity instead of position and presents a GPU oriented framework for falling leaves.

5.1 Motion modeling

Falling motion can be divided into six primitive motions as stated in [33–34]. These are steady descent (SD), periodic tumbling (PT), transitional chaotic (TC), periodic fluttering (PF), transitional helical (TH), and periodic spiral (PS). The falling motion is combination of one or more of these motions. You can see figure 5.1. In the following text I will describe how are these motions generated. I began testing everything in Matlab and also generated everything in it.

I choose Matlab because of the support of various high level mathematical functions (ODE solvers to be precise), support of graphical output and I have some experience using it. Thanks to it I could try my solutions fast and generate trajectory files that I saved to disk. And that are later loaded in C++.

5.1.1 Trajectory generation

Steady Descent (SD) can be seen as straight line going downwards which can be easily computed as:

$$x_t = 0 \tag{1}$$
$$y_t = -Ut \tag{2}$$

where U is the average descent velocity and t is the given time of the fall. The generated line by itself is not very good approximation of leaf falling straight down

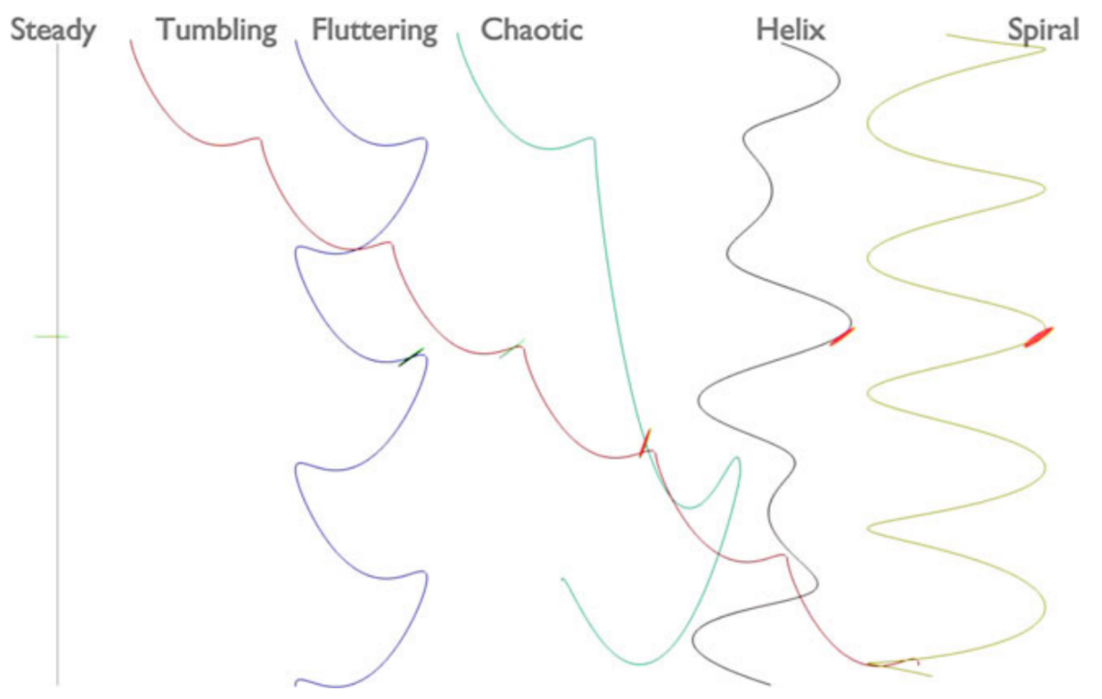


Figure 5.1. Primitive motions in x-y plane (2D).

since, some small movements in the other directions are happening. So I came with a bit better solution where fall I add a small random number in the other two directions (x,z) to achieve a small variation. The new equations are:

$$x_t = 0.01 * rand() \tag{3}$$
$$y_t = -Ut \tag{4}$$
$$z_t = 0.01 * rand() \tag{5}$$

The resulting fall trajectory you can see in figure 5.2.

Periodic Tumbling (PT), ***Transitional Chaotic (TC)*** and ***Periodic Fluttering (PF)*** all share the same equation for generation their fall trajectories. All of them are also 2D motions and can be created by combining similar trajectory motions. The equations are:

$$x_t = x_0 - \frac{A_x}{\Omega} sin(\Omega t) \tag{6}$$
$$y_t = y_0 - Ut - \frac{A_y}{2\Omega} cos(2\Omega t) \tag{7}$$

where A_x is the amplitude of vertical velocity, A_y is the amplitude of horizontal velocity. Ω is the angular frequency of the falling motion. x_0 and y_0 are starting points of the fall, these are not rely necessary and are here mainly to position the falling trajectory for the visualisation, in the generation it self they are both set to 0.

The tree main parameters that change the trajectory are A_x, A_y and U. A_x changes how much the leaf moves in x direction, the higher the A_x the higher the $[-x, x]$ range. A_y changes how much we are bending the ends of the fall (how much we swing back in the fall), the higher it is more we go up. U defines the speed of fall, the higher it is the more distance we fall in the given time t.

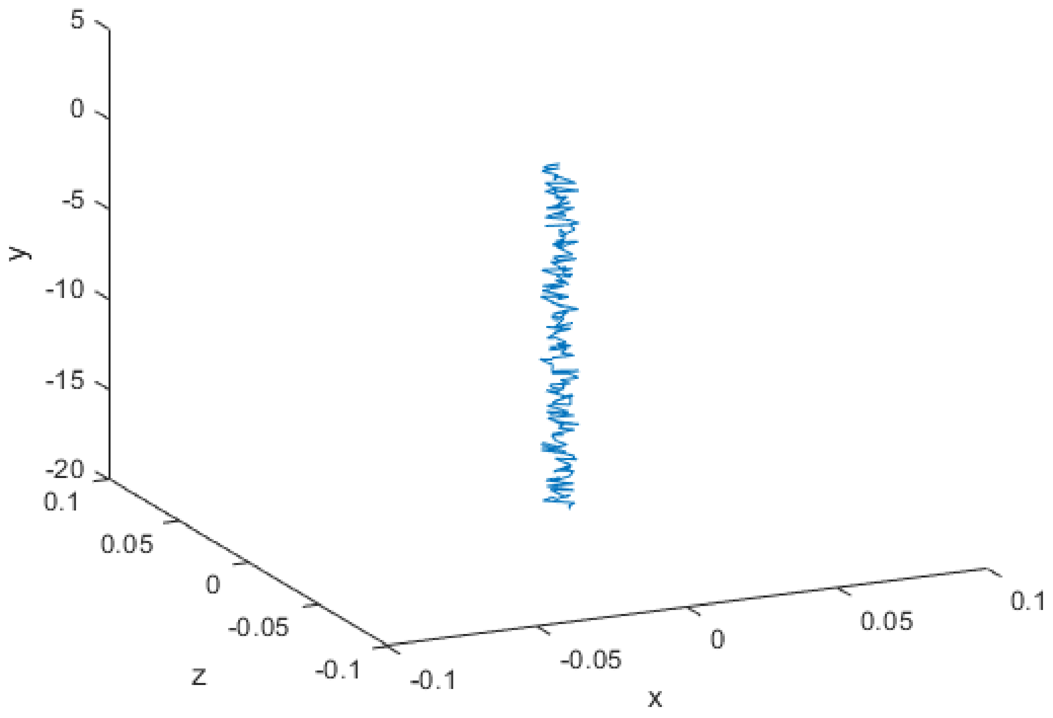


Figure 5.2. Steady Descend fall trajectory in 3D.

Originally in [9] they created a table of primitive motions by varying the A_x, A_y and U parameters and then combine them into the trajectory. If you want to check if you get similar results in the paper there is a table even with specific values that they used. In my implementation however I opted to not generate a whole set of these. I created separate Matlab files for each of the motions and a tweak the parameters for each segment myself until I was satisfied with the trajectory. This gave me more of a control of the whole trajectory and hence created better results. You can see the results in figure 5.3

The last two motions ***Transitional Helical (TH)*** and ***Periodic Spiral (PS)*** are 3D motions by design. TH represents the fall of a leaf as it would follow a petal trajectory and PS represents the fall of a leaf as it would follow a circular motion. Both of these motions are observed from the data gathered in [33]. To get the idea how it looks you can see in figure 5.4.

Both TH and PS use the same set of equation to describe their fall trajectory, these are:

$$x_t = A_e cos(\Omega t)(1 + E_e sin(k\Omega t) \tag{8}$$

$$y_t = h - Ut \tag{9}$$

$$z_t = A_e sin(\Omega t)(1 + E_e sin(k\Omega t) \tag{10}$$

where A_e is the amplitude of the elliptical oscillation generated in the $x - y$ plane, E_e is the ration of the minor axis to the major axis of the oscillation ellipse, k is the ratio of the period of elliptical oscillation to that of rotation of the falling object and h is the height from where the fall started.

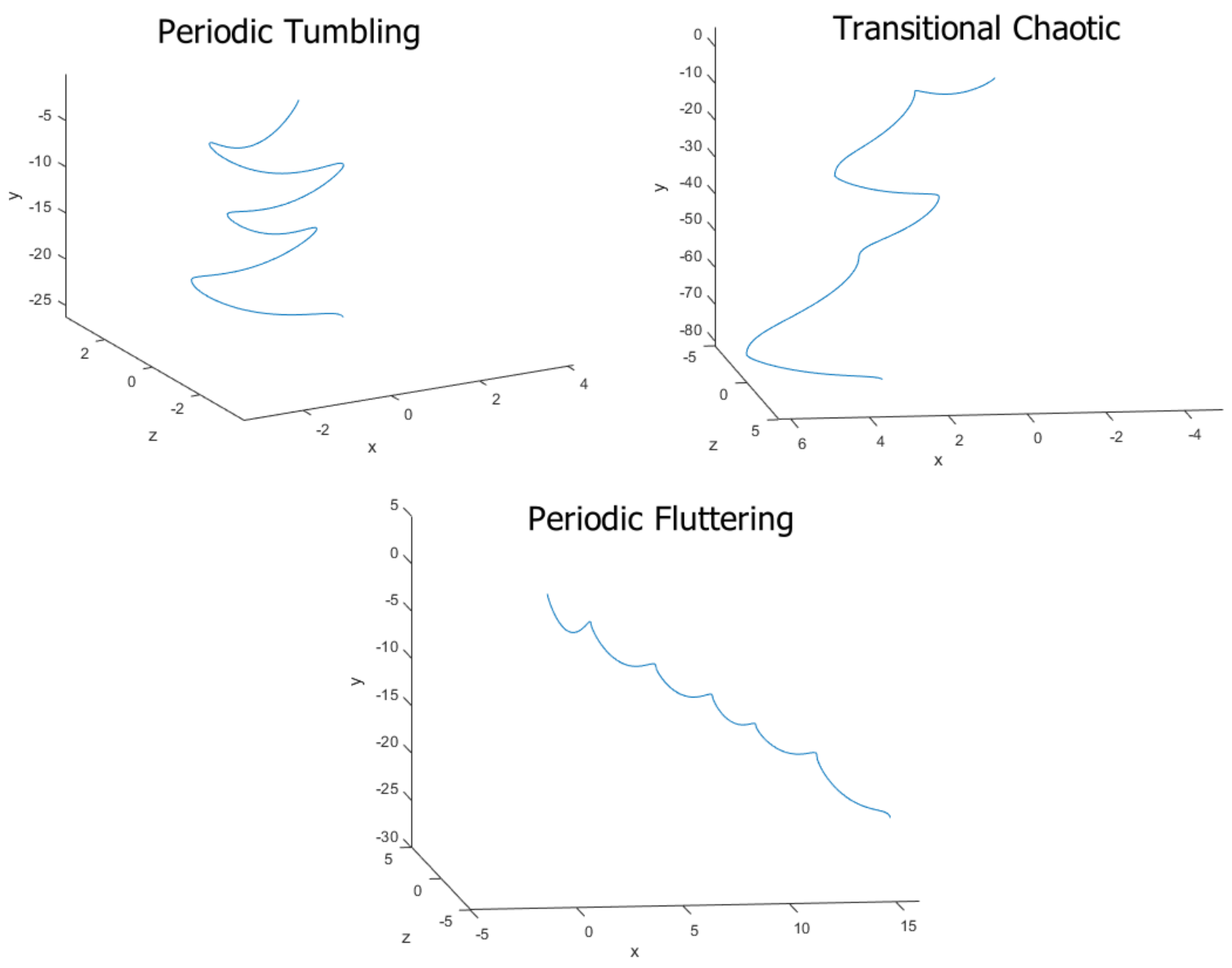


Figure 5.3. PT, TC and PF trajectory motions in 3D.

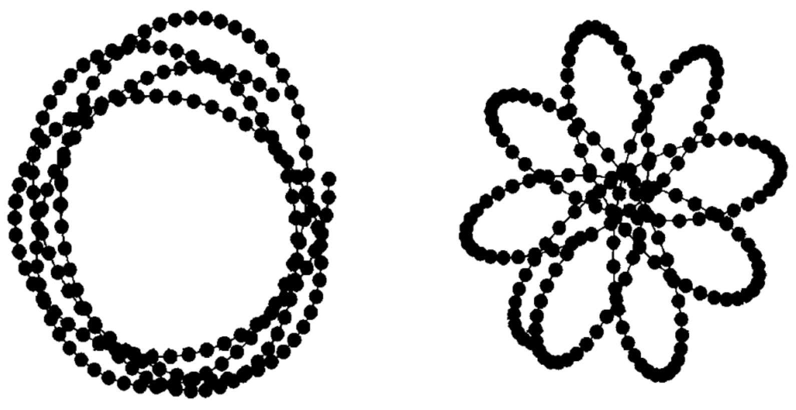


Figure 5.4. Top view of PS (left) and TH (right).

From the user perspective E_e assures the connectivity between petals, how acute the transition is (it can even create a smaller petal leaves next to the main ones if the values is big enough). k defines the number of petal leaves. A_e defines the width and height of the motion. To get a TH and PS we use these values:

$$TH \Rightarrow E_e \simeq 1, \quad k = 1 \tag{11}$$
$$PS \Rightarrow E_e \simeq 0, \quad k = 4 \tag{12}$$

You can see the results in figure 5.5.

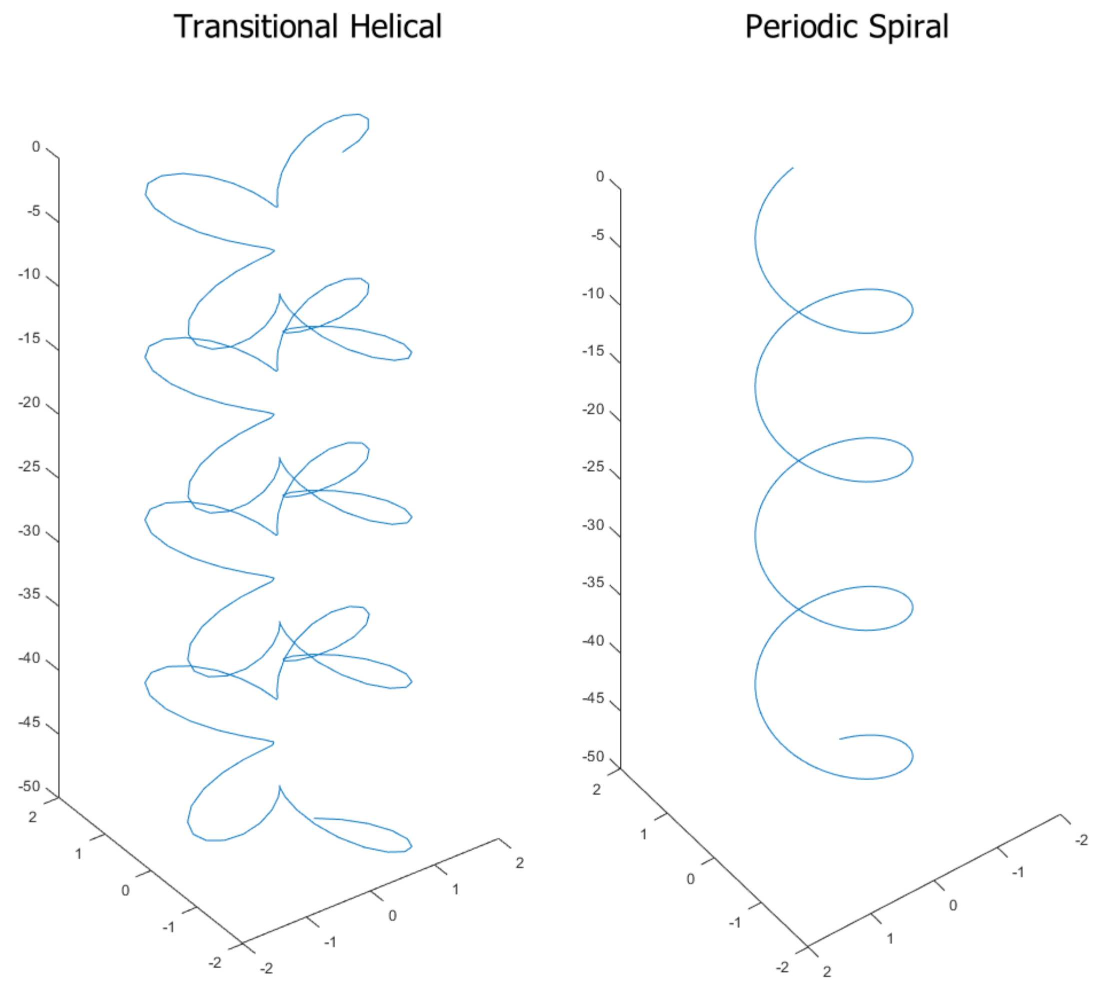


Figure 5.5. TH (left) and PS (right) trajectory motions.

With the given equation we can compute all the positions of the fall trajectory, but saving position leads to complications down the line. We would have to always transform the leaf to given position using matrix a better way to represent the change of position is through velocity. So instead of saving position we save velocity which can be easily computed since we know the position based on time and hence a simple subtraction will suffice (I used $diff()$ in Matlab).

5.1.2 Rotation

To achieve more realism having position change is not enough we also need rotation or more specifically angular velocity (rotation change). To compute the the angular velocity we will use ordinary differential equation (ODE) from [35], which is:

$$\frac{d\omega}{dt} = -k_a\omega - 3\pi\rho V^2 cos(\alpha+\theta) sin(\alpha+\theta) \tag{13}$$

where ω is the angular velocity of the leaf, ρ is the density of the leaf, θ is the angle with x-y plane and α is the angle with x-z plane. V is the velocity at the given time. To get a better understanding about the variables refer to figure 5.6.

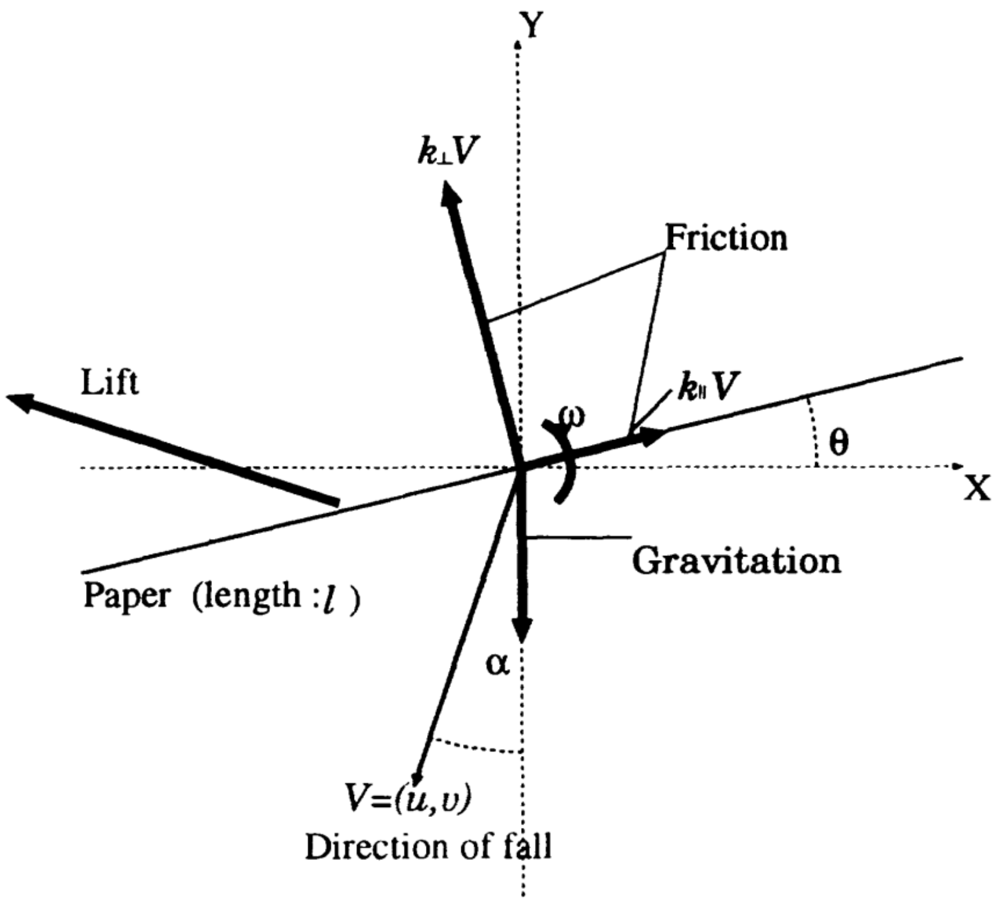


Figure 5.6. Schematic ilustration of the forces on the leaf. Image taken from [36].

To solve the second order ODE (we are deriving ω which is already a derivation $\omega = \frac{d\theta}{dt}$) I used Matlab function $ode45$, but before we can use this function we need to change the ODE to first order function. To do this we use the substitution trick where we introduce two new variables θ_1 and θ_2 where:

$$\theta = \theta_1 \tag{14}$$

$$\omega = \frac{d\theta}{dt} = \frac{d\theta_1}{dt} = \theta_2 \tag{15}$$

Having θ_1 and θ_2 we can substitute them into the ODE and get two new ones:

$$\frac{d\theta_1}{dt} = \theta_2 \tag{16}$$

$$\frac{d\theta_2}{dt} = -k_a\theta_2 - 3\pi\rho V^2 cos(\alpha + \theta_1)sin(\alpha + \theta_1) \tag{17}$$

Now we can use the Matlab $ode45()$. Only one thing remains is how to compute the velocity for the ODEs. All the other variables are either constants (π, ρ) or are the main variables that $ode45()$ computes it self. There is another problem that $ode45()$ uses different time step inside its implementation hence we cannot precompute the velocity and pass it as an argument. Luckily we know how to compute position based on time, so we can compute the exact position at the time and also compute value a bit in the past (in my implementation it is t - 0.01) to get the previous position. Having that we can compute the velocity again as a subtraction of these two. Since we have 3 different ways of how to compute the position, tree different version of $ode45()$ are called. For reference you can see the matlab function that $ode45()$ calls in figure 5.7. It is for PT,TC and PF motions.

```
function [dtheta_dt] = rot(t, theta, x0, y0, Ax, Ay, U, OHM, step)
theta1 = theta(1);
theta2 = theta(2);

ka = 4;
p = 0.1;

t0 = t - step;
xt_prev = x0 - (Ax/OHM) *sin(OHM*t0);
yt_prev = y0 -U*t0 - (Ay/(2*OHM)) * cos(2*OHM*t0);

xt = x0 - (Ax/OHM) *sin(OHM*t);
yt = y0 -U*t - (Ay/(2*OHM)) * cos(2*OHM*t);

u = xt - xt_prev;
v = yt - yt_prev;

V = u*u + v*v;
alpha = atan(u/v);

beta = alpha + theta1;

dtheta1_dt = theta2;
dtheta2_dt = -ka * theta2 - 3 * 3.14 * p * V * V * cos(beta) * sin(beta);
dtheta_dt = [dtheta1_dt; dtheta2_dt];

end
```

Figure 5.7. PT,TC and PF ODE Call Function.

With the solve ODE we now also have the angular velocity. The angular velocity is around the z axis and hence make sense for the 2D, but for 3D it is a bit weird. I did not manage to find if in the [9] they used just this angular velocity or they used some of their own. My main problem was with the 3D motions of TH and PS where its not really stated how it is computed. In the end I used the same idea of just doing the rotation around the z axis, but this could be further investigated as in how to compute better. Nonetheless the resulting angular velocity is visualised in the figure 5.8.

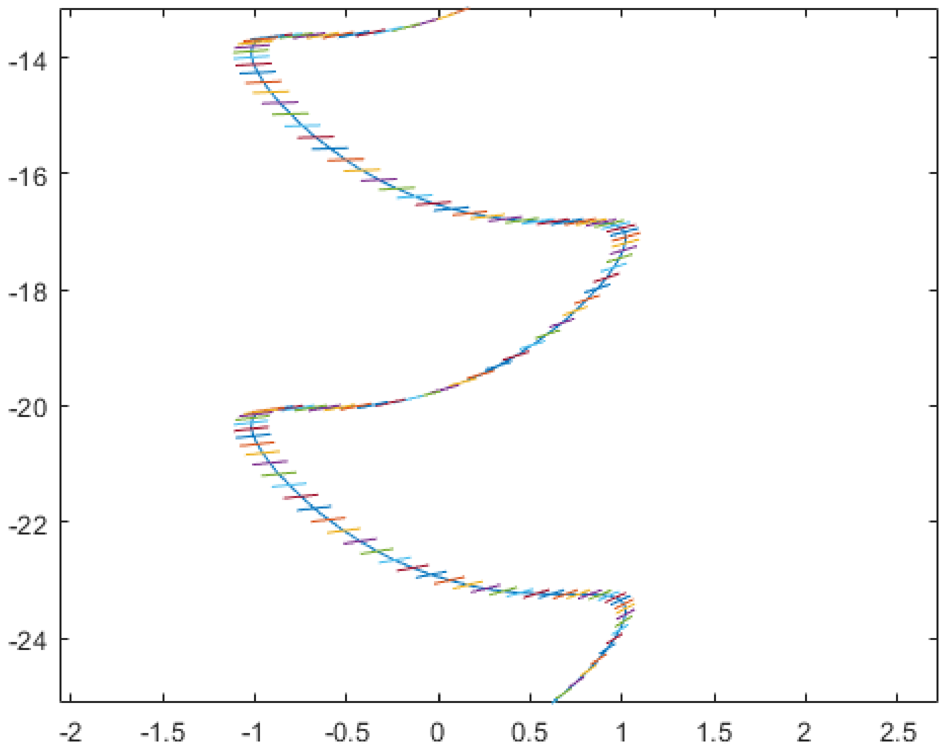


Figure 5.8. Angular velocity visualisation. The lines represent the leaf approximation.

5.2 Low Dimensional Fall Trajectory Representation

Now we have trajectories of all the primitive motions but to achieve enough variation we would have to create a lot of them for leaves to choose from. For GPU memory this is not efficient the less data the better. Luckily as stated in [9] we can just save a few basic ones and index them using a special low dimensional structure D.

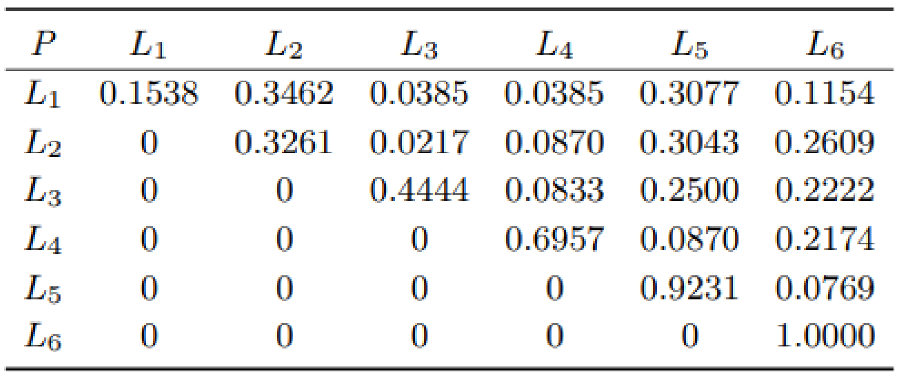
5.2.1 Load Motions From Disk

The motions are created in Matlab and saved into several .txt files on disk. The files are organized into a folder called Trajectories. In this folder there are files for each trajectory motion (SD,TH,TC, etc.) and a meta file. All of these files have specific names that correspond to each motion. SD trajectory motions is saved in SD.txt, TH trajectory motions is saved in TH.txt and so on.

The first file that we read is the meta file. Meta file contains an name of the motion (must be SD, PT, TC, PF, TH or PS) followed by the size of data on the next line. We read these two values and prepare vector to save the data to. Then we open another file based on the name given and read data from this file into the vector. Now we can create the motion based on the read values. We than read all of the data from meta file and load everything into vector of motions. There always must be all of the six motions present since the following algorithm depends on them (they sufficiently describe all possible fall motions).

5.2.2 Create Transitional Probability Table

Falling leaves don't follow the same motion of fall for the whole fall it generally changes between the states. For example the fall starts with SD trajectory and it changes next to TH for example. How do we determine when to change and what to change to. Luckily a table of probabilities is provided in [33]. The probabilities are based on real leaf falling and observations made by the researchers. You can see table in figure 5.9.



P	L_1	L_2	L_3	L_4	L_5	L_6
L_1	0.1538	0.3462	0.0385	0.0385	0.3077	0.1154
L_2	0	0.3261	0.0217	0.0870	0.3043	0.2609
L_3	0	0	0.4444	0.0833	0.2500	0.2222
L_4	0	0	0	0.6957	0.0870	0.2174
L_5	0	0	0	0	0.9231	0.0769
L_6	0	0	0	0	0	1.0000

Figure 5.9. Transition Probability Table. Picture taken from [33].

The probability table is created from a Markov Chain. Each row represents one state and each column represents the probability that the state will change into the other (or stay the same of we are on the diagonal line). It is also necessary to note that we can move one way and never back. This is based on the hypothesis from [33] where they qualitatively observed this property. Because of that the cells under the diagonal are all zero.

5.2.3 Create Trajectories D

Having loaded the data and having the probability table we can finally create the D structure. The creation is pretty straight forward. There are at most four switches in the tree fall, this is based on [9] where they claim that this is enough (but one can create more if he wants to). The feature D is defined as $D = S_1, S_2, S_3, S_4, S_5$, it consist of five states (we have five state because of the four switches) where each state is $S_i = L_i, T_i, P_i$. L_i is the primitive motion index, T_i denotes the total time for this motion stage, and P_i denotes the starting point for the primitive motion (as defined in [9]).

In code the value for L_i is taken from the Transition Probability Table, where we based on the current state choose the row of the array and based on a random number we chose the column (we generate a random number between 0 and 1 and subtract the probabilities form the row from it once the number is bellow 0 we return that state). P_i and T_i are just two generated random numbers but they do have to follow some conditions:

$$0 \leq P_i \leq 1 \tag{18}$$

$$0 \leq T_i \leq 1 \tag{19}$$

$$T_1 + T_2 + T_3 + T_4 + T_5 = 1 \tag{20}$$

to assure the last one that the sum of all times should equal to 1 is achieved by summing all of the numbers saved in the states and then dividing it by the sum. With that we get feature trajectory D. But we want to have several trajectories like this and all of the ma are generated the same way as I describe it. Thanks to the guided randomness we get a high variance of D from a small base of data. An visualisation of an example of how D motion looks like can be seen in figure 5.10.

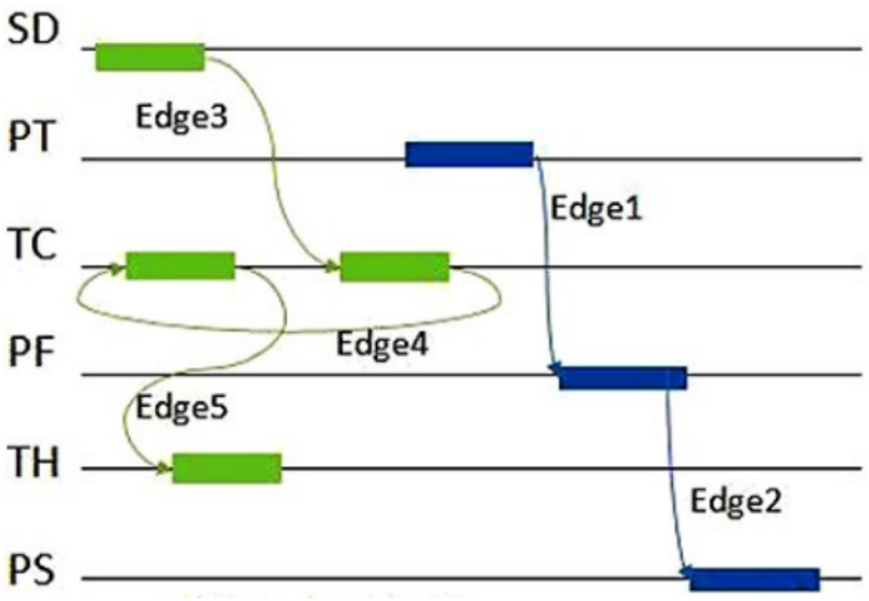


Figure 5.10. Falling leaf motion examples. For the green example, $M_1 = L_1, L_3, L_3, L_5$ for the blue example, $M_2 = L_2, L_4, L_6$. Picture taken from [9].

5.3 Leaf Class

To have the leaf data nice and tidy we create a Leaf class. The leaf class hold data for rendering these are VAO, VBO, InstanceVBO, texture id. VAO and VBO are indexes of vertex array object and vertex buffer object for OpenGL. InstanceVBO is a second VBO of the leaf and it represent per instance data for the leaf. This is a part of instance rendering and is discussed bellow. Last thing is texture id, which simply hold an index where the texture is saved in OpenGL GPU memory.

The vertex data for each leaf class are allocated on the CPU side only when the leaf is created since we don't change them. The vertices are even are same for all the leaves. They describe a quad around the origin. Each vertex also contains position, normal and uv coordinate information. The texture is loaded using OpenCV function $cv::imread(path, cv::IMREAD\_UNCHANGED)$ and then flipped because OpenCV uses different data layout.

Leaf class also holds additional information for trajectory selection. It holds an Did, fall start and posrot. Did corresponds to index of the given D structure. Fall start is the time when the leaf started it fall, this is used to get the proper time t between $[0, 1]$ (we need to know when we started the fall to know when to end it). Posrot is $glm::vec4$

attribute that holds center position of the leaf ($vec.xyz$) and rotation around the z axis ($vec.a$). Actually Did, fall start and posrot are saved in vectors and not simple types. This is because we use instancing and each vector column represents the data for this instance. You can see the leaf data in figure 5.11.

```
class Leaf {
  GLuint VAO;
  GLuint VBO;
  GLuint instanceVBO;
  GLuint texture;

  std::vector<int> Dids;
  std::vector<float> fall_starts;
  std::vector<glm::vec4> posrot;
}
```

Figure 5.11. Leaf class.

5.3.1 Leaf Generation

Setting up a lot of instanced leaf data manually would be very inconvenient so a simple leaf spawner is set up. The spawner represents a quad area in space in the x-z plane (note OpenGL uses Right-handed system) with given height in y axis. With these bounds we now know where we can spawn the leaves and setting the other instance data. The process is pretty simple and is described in function $GenerateLeafInstance(float \quad time)$ in figure 5.12. Using this we can generate any amount of leaf instances with ease.

```
GenerateLeafInstance(float time)
{
  int Did = std::floor(GetRandom() * Ds.size());
  float fall_start = time;
  vec4 start = vec4(GetRandom()*spawnerXsize - spawnerXsize/2.0f,
                    spawnerZ,
                    GetRandom() * spawnerYsize - spawnerYsize / 2.0f,
                    0);
  AddInstance(Did, fall_start, start);
}
```

Figure 5.12. Leaf generation.

5.4 Instanced rendering

Instance rendering is an efficient way how to render models that contain the same vertex and only differ in some small aspect. A classical example is a meteoroid field around a planet (if you google instanced rendering it will be most like the example that will be

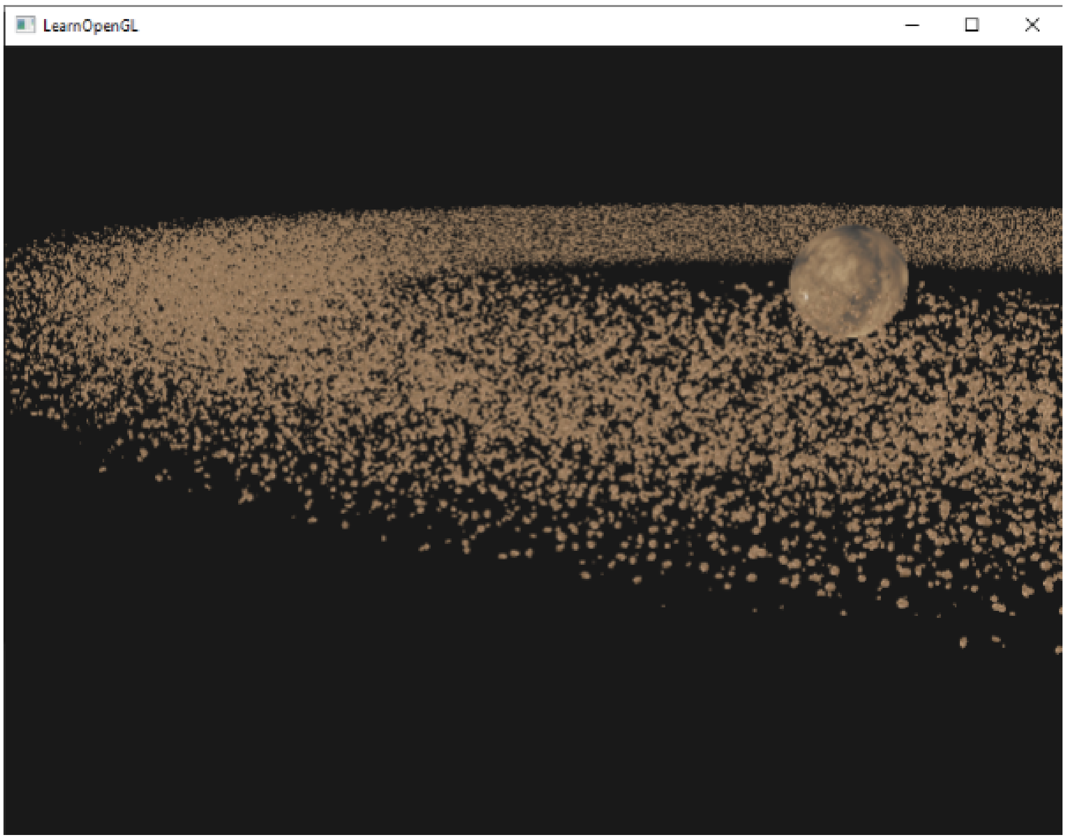


Figure 5.13. Instancing Example. Image taken from [26].

used. In our case it basically the same thing only we have leaves instead of meteoroids. This inspiration and know how was taken from the learnopengl website [26] chapter Instancing.

It also should be stated that in the main paper [9] that this chapter is based on they didn't use instanced rendering instead they choose geometry shader in which they generated the quad based on a center value, tangent and sizes in x and y direction. I opted to use instanced rendering because I believe its better suited for this task even more than geometry shader, since we will reduce the number of render calls on GPU.

As stated in section about Leaf above we have the same geometry for a leaf that consists of VAO and VBO. We set for the VBO how the data in VBO is organized and how it should be parsed in vertex shader on the GPU. For instance rendering we will do something similar with one key difference. We will create an new VBO called instanceVBO and also set how the data should be parsed in vertex shader. But we also need to set use opengl function $glVertexAttribDivisor(3, 1)$ that specifies that the attribute on the position 3 should be set only once per rendering instance. This means that the data is the same for the whole instance. The code sniped for the attribute setting is in figure 5.14

The extra data for instance rendering are the data that we get from the update part of the render loop. These are the new updated positions and rotation for the leaf. One can understand it as an transform of the leaf vertex data that are originally at the center of the world, with this transform we will place the leaf in a correct place in the world. One could think so now we will create a transform 4 by 4 matrix and send it via the VBO that we prepared. But one can also deduce that the VBO is parsed as 4 floats.

```
//instanced data
glEnableVertexAttribArray(3);
// this attribute comes from a different vertex buffer
glBindBuffer(GL_ARRAY_BUFFER, instanceVBO);
glVertexAttribPointer(3,4,GL_FLOAT,GL_FALSE,4*sizeof(float),(void*)0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
// tell OpenGL this is an instanced vertex attribute.
glVertexAttribDivisor(3, 1);
```

Figure 5.14. Code for setting instanced attribute for instance rendering.

Since we only compute rotation around the z axis and translation we just need the four values and the actual transformation is computed and applied in vertex shader.

Now the GPU know how to parse the data all we need to do now is send them to the GPU. First we send the data to the VBO we do this in every frame since the data changes every frame. For the actual change of data we I use the function glBufferData. This is not efficient approach but a simple one. One could create a big buffer for all the data and with the usage of glBufferSubData only change part of the data. Also some logic as to how many leaves should be rendered based on the shorter data should be considered. I will leave this as an potential small extension of this work, to make it more efficient.

The instance rendering function to call is pretty similar to a normal call, since all the VBO for the given VAO are updated we just need to set the texture of the object. The texture is set through binding at the texture unit with index 0. And finally a function $glDrawArraysInstanced(GL\_TRIANGLES, 0, 6, N)$ is called. It hase one more attribute that specifies the number of instances N, in our case its the number of leaves. The call for the render function can be seen in figure 5.15

```
void Draw() {
  glActiveTexture(GL_TEXTURE0);
  glBindTexture(GL_TEXTURE_2D, texture);

  glBindVertexArray(VAO);
  glDrawArraysInstanced(GL_TRIANGLES, 0, 6, posrot.size());
}
```

Figure 5.15. Leaf render function.

It should be also stated that the type of leaves are distinguished by the texture that they use. So if in a scene is a leaf that has a given texture it will be rendered through one instanced render call. This due to the fact that I did not find an easy way or way at all to somehow index texture based on a value send to each instance. That being said few more render calls for each of the unique texture will not hinge the performance too much if at all.

5.5 Render

The overall render algorithm can be summarized in few steps, I describe it in figure 5.16 using a pseudo code function that have names according to what they do. Most of the function were described previously already (LoadMotions, GenerateTrajectoriesD, etc.). But seeing them in order should give you a good idea of how it works. As with most render programs first there is the preparation phase where we setup and initialize the data. Then we proceed to into the render loop where we update, set the update to the GPU and send the draw call to the GPU.

```
InitOpenGL();
LoadMotions();
GenerateTrajectoriesD();
GenerateLeaves();
PreparedMotionsAndDsForCUDA();
while(){
  t = GetTime();
  UpdateLeafInstacesCUDA(t);
  SetUniformsAndInstanceVBOToOpenGL();
  Draw();
}
```

Figure 5.16. Leaf render overview.

There are still few parts that are not discussed. I will not go into the specific of how to initialize OpenGL since it is not the focus of this thesis. But I will describe the remaining update step and how the Motions and Ds are prepared for CUDA since these the only ones left that were not touched up on. I will also describe the vertex and fragmen shaders that are used.

5.5.1 Prepare motions and Ds for CUDA

To send data to the GPU using CUDA we need to simplify the data a bit. CUDA is more user friendly than OpenGL (compute shader vise) but it is still has a long way to go to have the same levels as C++. The main problem is sending data to the GPU. We need to use cudaMalloc and cudaCopy to get the data over to the CUDA GPU memory. Also this data has to be linear (1 dimensional) since the cudaCopy takes the pointer address and the amount of data and then it copies this chunk of memory to the destination pointer. But we do have 2D data (for example each D has a vector of several states). To address this issue I linearized the vector into 1D array and created additional array with meta information. You can see the meta class in figure 5.17.

The meta information consist of the startIdx and size, startIdx points to where the given D (for example) starts and size how many states it has. We combine this with the linearized 1D vector and get the given states for given Did that is saved for the instance. This way we can send the data to the CUDA GPU memory and access it in the kernel. We do this for D and motions.

```
class Meta{
  int startIdx;
  int size;
}
```

Figure 5.17. Meta class.

5.5.2 Leaf Update

To update the leaf instance data we call a function $SimulateLeafFallCUDA$ from C++. This function is a wrapper function to get to the CUDA side implementation (same process as in chapter Tree Simulation). In the update we index the given trajectory based on given D and time. Also we simulate a simple wind effect given by direction and wind strength.

First in the update we need to sent the data to the GPU, we do that allocation space using $cudaMalloc$ and then copying the data using $cudaCopy$. We do this for Dids, fall, posrot, Ds, DsMeta, motionsMeta. These are instance ids to the structure D, when the fall started, current position and rotation (around the y axis) of the leaf, array of structure D, meta data for structure D, array of motions and meta data for motions respectively. For simple types we don't need to allocate space manually, these are timePassed, transScale, rotScale, windspeed and windDir. They are time passed from the start of the simulation, scale of the computed translation, scale of the computed rotation, speed of the wind times delta time and the direction of the wind.

All necessary data are loaded into the kernel so we can start it. We follow the paper [9]. First we get the ID of the kernel, this ID is the same as the ID of the instance. Using this ID we get the Did, fall_start and posrot form the corresponding arrays. We compute the time t of the fall by $t = timePassed - fall\_start$

Then we sample the structure D, we do this using the $SampleD(Did, Ds, DsMeta, t)$ function. To get the current state of the fall based on the given time we iterate through all of the states of the given Did (based on the data from meta file). Each state holds T its time duration, we subtract this value from the current t and check if the t is negative. Because if the t is negative we found the state we are currently in. To get the time in the motion trajectory we compute: $time = t_orig - t_s + s.P$. $t_orig - t_s$ will give us the time in the state and $s.P$ will position us in trajectory function. If the time t is bigger than 1 subtract 1 from it and index the motion in circular fashion. Since we have t of the motion, we will return the motion ID and the time. If the time of the motions exceeds its duration (it ends) we will return $[-1, -1]$ to indicate that we should not apply any change. You can see the function in pseudocode in figure 5.18

Now that we know the motion and time in the motion we can sample it. For this we use the $SampleMotion(motions, motionsMeta, L, t)$ function. In this function we first get the meta data of where to index. Then we get the idx by calculating $idx = start + t * size$ with which we get the position in the array based on time t. Since we will get the idx value in floating point representation we need to floor to get just the integer. Also to achieve more accurate result we linery interpolate between the index value and the next to get the proper value (this is done because the step for creating the trajectory is not necessarily the same as we index it in the simulation). You can see the function in figure 5.19.

```
__device__ DSample SampleD(Did,Ds,DsMeta,t) {
  meta = DsMeta[Did];
  int start = meta.startIdx;
  int size = meta.size;
  float t_orig = t, t_s = 0;
  for (int i = 0; i < size; i++) {
    DState &s = Ds[start + i];
    t -= s.T;
    if (t <= 0) {
      float time = t_orig - t_s + s.P;
      if (time > 1.0f)
        time -= 1.0f;
      return [s.L, time];
    }
    t_s += s.T;
  }
  return [-1, -1];
}
```

Figure 5.18. Sample D algorithm.

```
__device__ vec4 SampleMotion(motions,motionsMeta,L,t) {
  int start = motionsMeta[L].startIdx;
  int size = motionsMeta[L].size;
  float idx = start + t * size;
  int first = floor(idx);
  int second = first + 1;
  if (second > start + size) {
    return glm::vec4(0, 0, 0, 0);
  }
  return MyLerp(motions[first], motions[second], idx - first);
}
```

Figure 5.19. Sample motion algorithm.

With the change known we can apply it to the posrot of the instanced leaves. But first we scale it with user defined values transScale and rotScale. Here we also apply the wind on the the leaf. We save these values into the given posrot (posrot[ID]). The code for this is in figure 5.20. After the kernel ends (we wait for it using $cudaSynchronize()$) we copy the values into the CPU array of the leaf.

To propagete the change to GPU we update the instanceVBO data based on we call a leaf function $SetInstanceData()$ where we copy the new data to the buffer using $glBufferData$. Having that we can now call the $glDrawArraysInstanced$ on the leaf. We do this process for all unique leaves, in our simple case these are the leaves that

```
change.x = change.x * transScale + windDir.x * windSpeed;
change.y = change.y * transScale + windDir.y * windSpeed;
change.z = change.z * transScale + windDir.z * windSpeed;

change.a = change.a * rotScale;

posrots[ID] = posrot + change;
```

Figure 5.20. Apply change code.

have different texture. But if one would extend the current solution a different geometry could also diverse the leaves.

5.5.3 Shaders

We apply the set the position and rotation in vertex shader. In previuos section I have described how it is created and how it is send to OpenGL memory. Vertex shader takes the data and applies it to the vertex. I have create two function to help me apply the changes, these are translate and rotate. You can see them in figure 5.21. They translate and rotate the vector accordingly. The rotation is in around the z axis. I opted to do this inside the shader itself instead of creating an matrix to represent the transformation because it has less operations and need less memory space.

```
vec3 translate(vec3 vec, vec3 trans){
   return vec + trans;
}

vec3 rotate(vec3 vec, float rotz){
   float x = cos(rotz)* vec.x - sin(rotz) * vec.y;
   float y = sin(rotz)* vec.x + cos(rotz) * vec.y;

   return vec3(x,y,vec.z);
}
```

Figure 5.21. Change functions.

The rest of the vertex shader is pretty straight forward and does things same as you would do in a normal scenario. You can see the vertex shader in figure 5.22

Fragment shader is also pretty normal. We have a texture that we load using function *texture*(). This texture has an alpha channel and mask it using it. We test if it does have an alpha equal to zero and if so discard this fragment. Using normal and view direction we than compute the color. Since we have only diffuse texture we use ambient color as an multiplication of said texture times 0.2. Specular color is ignored, leaf will do not reflect light. You can see the shader in figure 5.23.

```
#version 430
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNorm;
layout (location = 2) in vec2 aTexCoords;
layout (location = 3) in vec4 aOffset;
layout (location = 0) uniform mat4 model;
layout (location = 1) uniform mat4 view;
layout (location = 2) uniform mat4 proj;
layout (location = 3) uniform mat3 normalModel;
out vec3 Normal;
out vec3 FragPos;
out vec2 TexCoords;

void main()
{
   vec3 instancedPos = translate( rotate(aPos,aOffset.a), aOffset.xyz );
   gl_Position = proj * view * model * vec4(instancedPos, 1.0f);
   FragPos = vec3(model * vec4(instancedPos,1.0f));
   Normal = rotate(aNorm, aOffset.a);
   TexCoords = aTexCoords;
}
```

Figure 5.22. Vertex shader.

```
#version 430
out vec4 FragColor;
in vec3 FragPos;
in vec3 Normal;
in vec2 TexCoords;
uniform vec3 viewPos;
layout (binding = 0) uniform sampler2D image;

void main()
{
   vec4 diffColor = texture(image, TexCoords);
   if(diffColor.a == 0)
      discard;
   vec3 norm = normalize(Normal);
   vec3 viewDir = normalize(viewPos - FragPos);
   vec3 color = CalcDirLight(dirLight, norm, viewDir, diffColor.rgb);
    FragColor = vec4(color, diffColor.a);
}
```

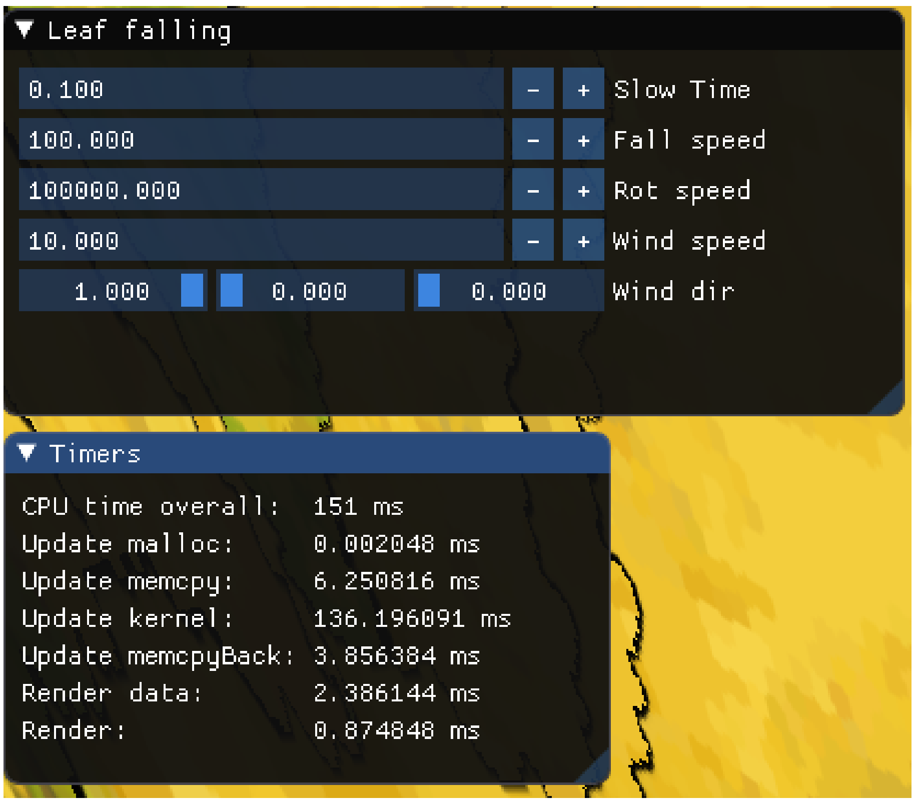
Figure 5.23. Fragment shader.

5.6 GUI

Using ImGui [37] I have create a simple GUI for the application. In the GUI you can set various variables. You can set how fast the time passes with SlowTime input box. The strength of fall and rotation change with Fall Speed and Rot speed input box. And you can set the wind speed and wind direction.

There is also a second part of the GUI where the timings of various parts of the algorithm are presented. To get the time itself I use query calls in OpenGL and use the event system in CUDA to do the same. This way is better and more accurate than timing the calls on CPU side since the communication offset and when we actually return from the calls might differ. I have also divided the measurement into sections to get a better under1standing what takes what time. And of course overall time on the CPU is also measured.

The GUI can be seen in figure 5.24.



```
▼ Leaf falling
0.100                  - +  Slow Time
100.000                - +  Fall speed
100000.000             - +  Rot speed
10.000                 - +  Wind speed
1.000      0.000      0.000  Wind dir

▼ Timers
CPU time overall:   151 ms
Update malloc:      0.002048 ms
Update memcpy:      6.250816 ms
Update kernel:      136.196091 ms
Update memcpyBack:  3.856384 ms
Render data:        2.386144 ms
Render:             0.874848 ms
```

Figure 5.24. User interface for the leaf falling simulation.

Chapter 6 Results

In this chapter I will present all the findings and results that I got from the implemented algorithms in a shorter and compact way. First the Leaf Acquisition results are presented followed by Tree Simulation and Leaf Falling Simulation results. Also a measurement of speed and memory requirement is presented.

6.1 Leaf Acquisition

The thesis begins by analysing various way of how to implement following task of this thesis. It outputs an table that summarizes what each analysed paper is solving. In this section also an general approach of how how the task will be solved, basically which papers I have chosen to implement.

Following is an chapter about Leaf Acquisition. This chapter focuses on generating leaf texture based on a shape of the image. First a way how to grow leaf veins semi-automatically is introduced. The result can be seen in figure 6.1. Then a process of generation low dimensional leaf texture color is described. The results of which can be seen in figure 6.2

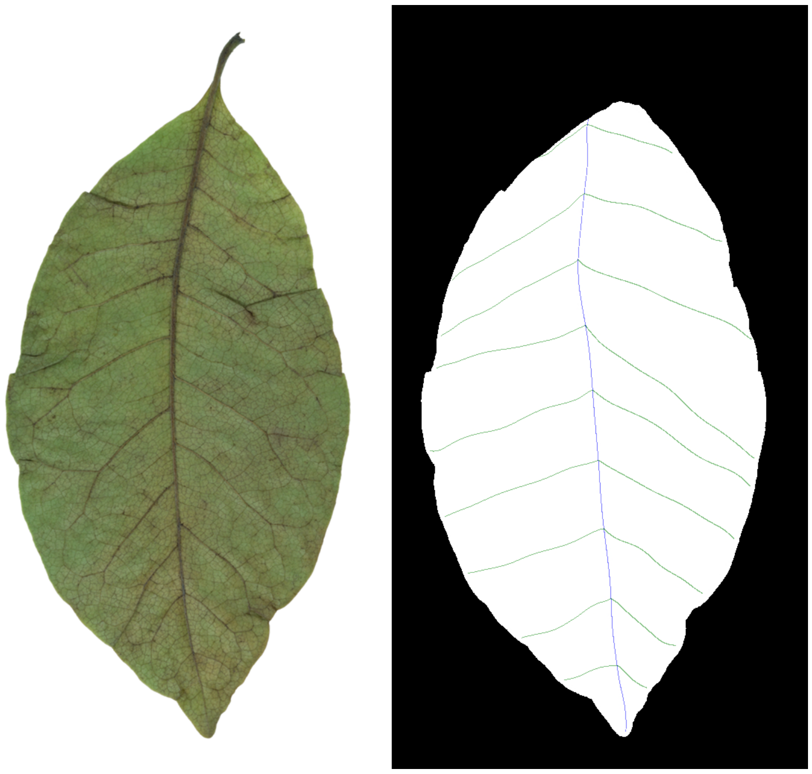


Figure 6.1. Leaf vein Generation output.

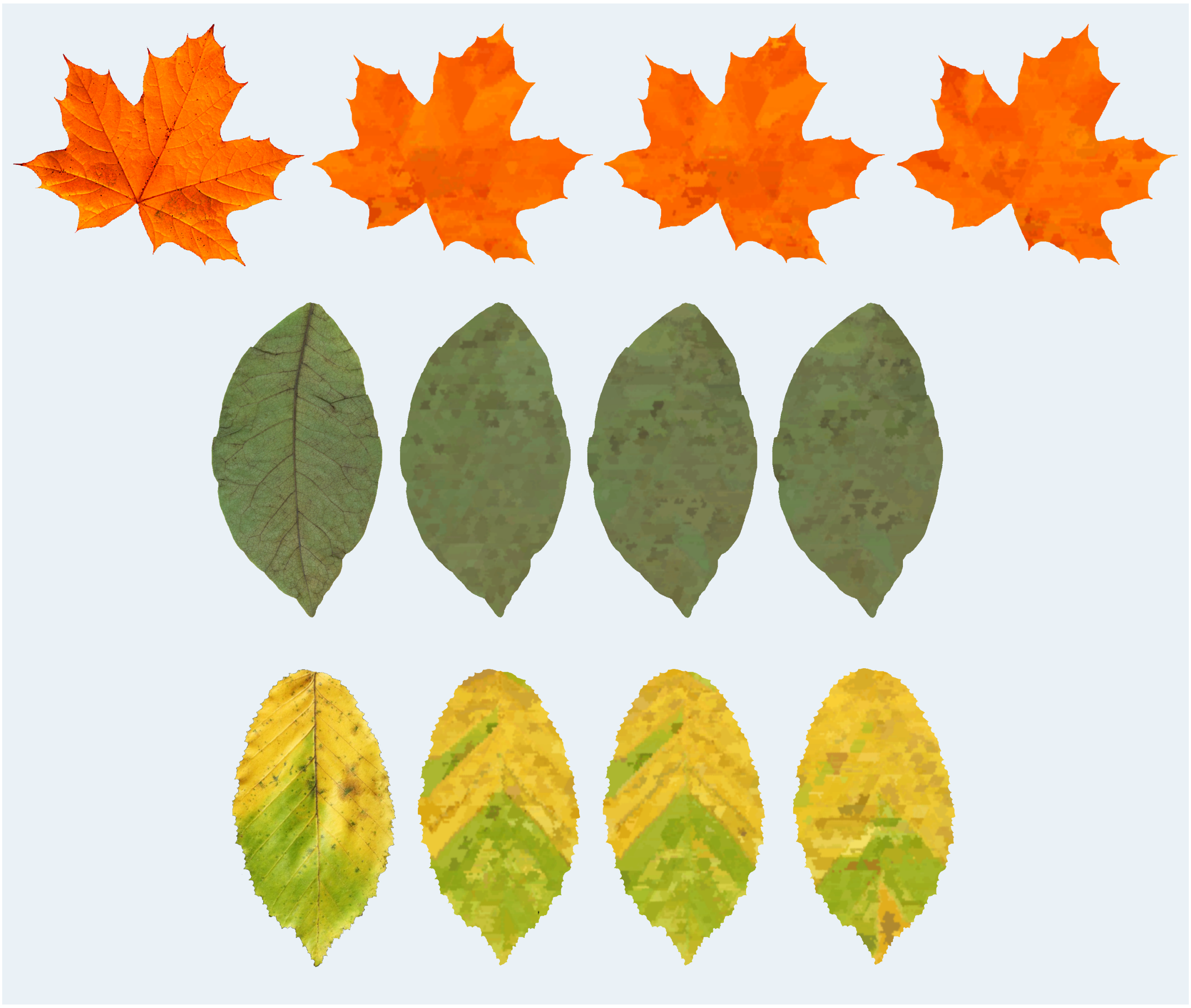


Figure 6.2. Comparison of generated textures. From left to right: original image, generated texture, generated texture with same regions as previous image and generated texture with different region selection.

As for the measurement for this part of the thesis first one has to realize that this is an offline method. So speed is not really the focus here. So it was not my focus either and it doesn't make sense to measure it precisely. Another thing is that to measure the speed of this part would be fairly tricky. It is highly dependent not only on the input images but also on the user input (leaf region selection, root mark). In most cases it runs within few minutes.

As for the memory requirements we need 3 input images and we generate k helper images (for example levels in the pyramid). This means that the memory is linearly dependent on these constants so we can abbreviate them. Another thing that we need is to save region data and this depends on user input, but still the regions are selected from the images so each image region representation can be seen as image of its own. This mean that we again add some constant amount of data. The memory requirements therefore are $O(N)$ linearly dependent on the amount of pixels.

6.2 Tree Simulation

The next chapter a simulation of tree dynamics is implemented. A specific FBX model creation is described to fit the algorithm. Also a potential way how to approximate physical properties from the specific FBX model is introduced. Then the simulation computation is described with a combination of math and code. But there is an issue with the simulation because the forces that would prevent the tree to bend unnaturally are not strong enough to counter the forces that are applied to the tree. This would require a further investigation. The result of the simulation can be seen in figure 4.15.

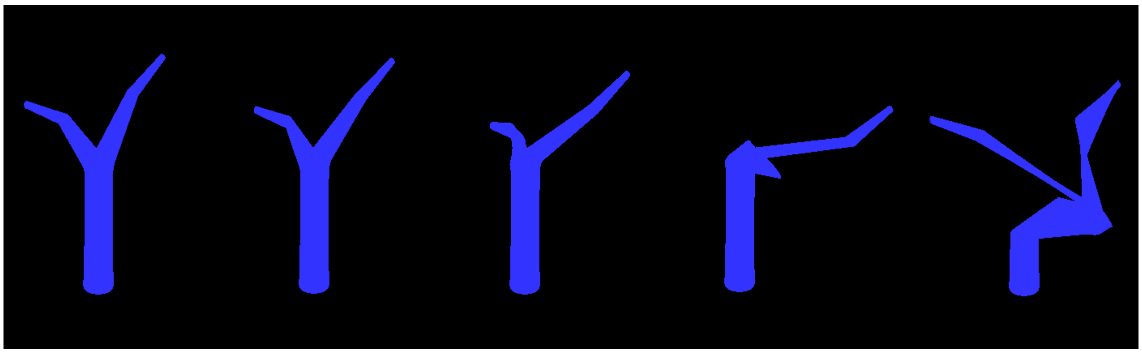


Figure 6.3. Tree simulation result. Time frames begins from left to right. You can see that the tree over rotates.

As for the measurement for this part since it is a failed implementation it doesn't really make sense to measure this part. Nonetheless we do achieve a real time fps. The main tree factor are render, Dynamics Calculation pass and Integration Of Movements pass. In render it self is straight forward, the only thing that we need to do is to send the bone transformation data to vertex shader. As it is right now we just use an uniform array. In this aspect one could use a different more efficient way. As for the Dynamics Calculation we call a CUDA kernel with one thread for each rigid body, this means that theoretically we can get $O(N/T)$ time. The Integration Of Movements is a bit tricky since we need to start it per level of the tree. Thanks to that we get $O(N/T + d)$.

Memory vise we each rigid body has several attributes but they are constant and not growing. So this means that the memory once its loaded doesn't change. So we get $O(N)$ memory requirements for each rigid body.

6.3 Leaf Falling Simulation

The next chapter describes a simulation of falling leaf using GPU. A way to compute the trajectory is introduced using Matlab. You can see the resulting curve in figure 6.4. Next a way how to represent these trajectories efficiently using low dimensional structure D on GPU is presented. Then the simulation computation is described using this structure D. You can see the result in figure 6.5. For video of the falling leaves refer to attached files.

The speed of the simulation is represented in table 6.1. You can see that for less than 100000 leaves the overall time is similar this suggests that the overhead off calling function on the GPU is larger than the algorithm itself. When we hit the 100000 mark the time starts to rise. As you can see in the table the times grow linearly. The measurement is divided into several sections in order of execution (first is Update

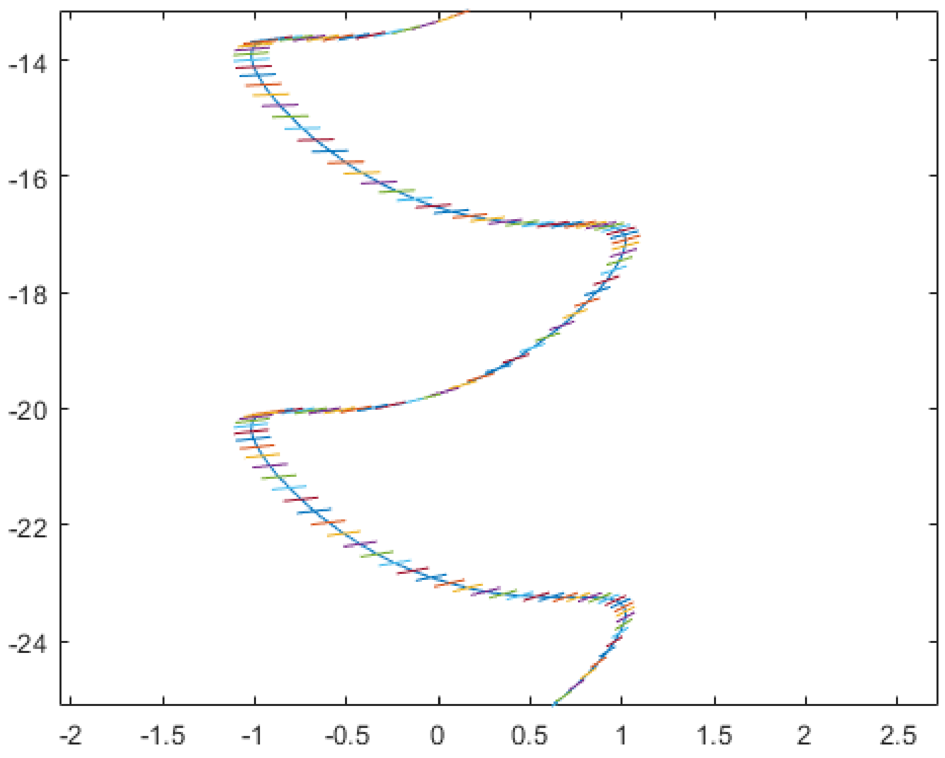


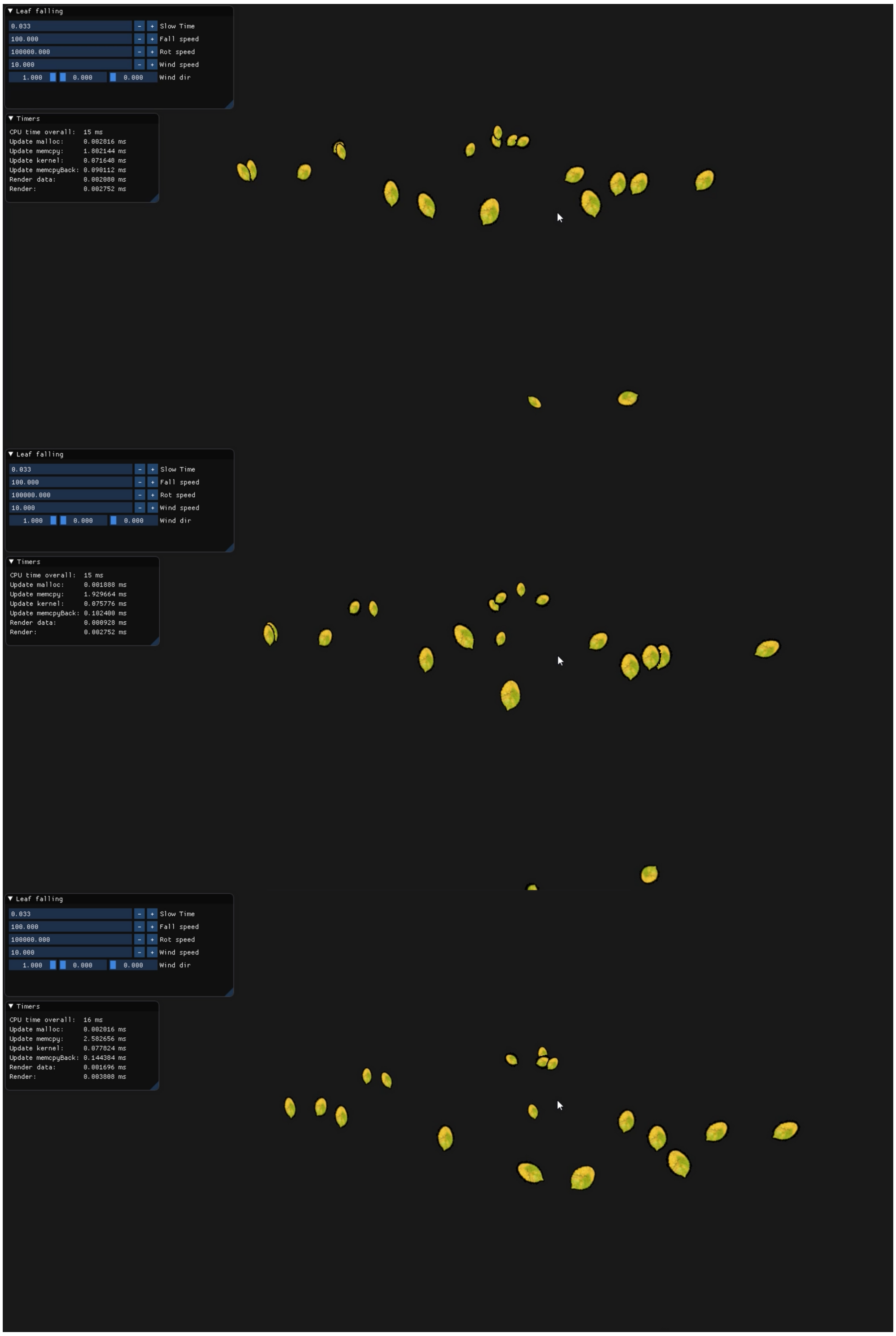
Figure 6.4. Computed trajectory result. The lines represent the leaf approximation.

malloc, then Update memcpy and so on) in which you can see that the kernel execution is by far the most expensive of all of them. As you can see that up to 100000 leaves we get an usable frame rate but after we add more leaves we enter the 'under 60 fps territory'. Memory requirements here are $O(N)$ we have data per leaf that does not expand during the algorithm.

number of leaves:	1000	10000	100000	500000	1000000
CPU overall	13 ms	14 ms	16 ms	73 ms	149 ms
Update malloc	0.002 ms	0.002 ms	0.002 ms	0.002 ms	0.003 ms
Update memcpy	0.383 ms	0.38 ms	1.012 ms	3.09 ms	5.80 ms
Update kernel	0.26 ms	2.18 ms	13.39 ms	65.644 ms	131.01 ms
Update memcpy back	0.1 ms	0.12 ms	0.41 ms	1.77 ms	3.596 ms
Render set data	0.003 ms	0.001 ms	0.14 ms	1.13 ms	2.386 ms
Render	0.008 ms	0.190 ms	0.22 ms	0.53 ms	0.921 ms

Table 6.1. Falling Simulation Speed.

It should also be mentioned that we measure the times on both GPU and CPU. For CPU we simply save two time stamps (start and stop of the timer recording) given by $glfwGetTime()$ function. With GPU we use CUDA and OpenGL so we use their constructs to save the time. For CUDA we use the $cudaEventRecord(event)$ to save two events before and after the wanted part of code that we want to record. To get the result we use the code from figure 6.6. Where we first synchronize the two events (wait for the



```
▼ Leaf falling
0.033          - + Slow Time
100.000        - + Fall speed
100000.000     - + Rot speed
10.000         - + Wind speed
1.000  0.000  0.000 Wind dir

▼ Timers
CPU time overall:  15 ms
Update malloc:     0.002816 ms
Update memcpy:     1.802144 ms
Update kernel:     0.071648 ms
Update memcpyBack: 0.090112 ms
Render data:       0.002080 ms
Render:            0.002752 ms
```

```
▼ Leaf falling
0.033          - + Slow Time
100.000        - + Fall speed
100000.000     - + Rot speed
10.000         - + Wind speed
1.000  0.000  0.000 Wind dir

▼ Timers
CPU time overall:  15 ms
Update malloc:     0.001888 ms
Update memcpy:     1.929664 ms
Update kernel:     0.075776 ms
Update memcpyBack: 0.102400 ms
Render data:       0.000928 ms
Render:            0.002752 ms
```

```
▼ Leaf falling
0.033          - + Slow Time
100.000        - + Fall speed
100000.000     - + Rot speed
10.000         - + Wind speed
1.000  0.000  0.000 Wind dir

▼ Timers
CPU time overall:  16 ms
Update malloc:     0.002016 ms
Update memcpy:     2.582656 ms
Update kernel:     0.077824 ms
Update memcpyBack: 0.144384 ms
Render data:       0.001696 ms
Render:            0.003808 ms
```

Figure 6.5. Example of the leaf fall simulation. Time passes from top to bottom.

stopto finish) and then using $cudaEventElapsedTime(time, start\_event, stop\_event)$ we get the elapsed time. For OpenGL we use an GL_TIME_ELAPSED query. We start it using glBeginQuery(GL_TIME_ELAPSED, query) and to stop it we use glEndQuery(GL_TIME_ELAPSED). To get the time we get use the glGetQueryObjectui64v(query, GL_QUERY_RESULT, time) function.

```
float time;
cudaEventSynchronize(stop_event);
cudaEventElapsedTime(&time, start_event, stop_event);
```

Figure 6.6. CUDA time measure.

Chapter 7
Conclusion

The main focus of this thesis is realization of real-time simulation of huge numbers of falling leaves using the GPU. It also describes a way how to create a base color texture color for rendered leaves. It analyses and tries to implement a tree dynamics simulation. Further text describes the chapters.

The thesis begins by analysing various way of how to implement following task of this thesis. It outputs an table that summarizes what each analysed paper is solving. In this section also an general approach of how how the task will be solved, basically which papers I have chosen to implement. This is not a complete analysis of all the papers discussing the thesis topic but just the ones were deemed relevant. One could expand this and make a proper survey.

Following is an chapter about Leaf Acquisition. This chapter focuses on generating leaf texture based on a shape of the image. First a way how to grow leaf veins semi-automatically is introduced. Then a process of generation low dimensional leaf texture color is described. The vein generation is not perfect and hand drawn veins are still better.

The next chapter a simulation of tree dynamics is implemented. A specific FBX model creation is described to fit the algorithm. Also a potential way how to approximate physical properties from the specific FBX model is introduced. Then the simulation computation is described with a combination of math and code. But there is an issue with the simulation because the forces that would prevent the tree to bend unnaturally are not strong enough to counter the forces that are applied to the tree. This would need a further investigation.

Then simulation of falling leaf using GPU. A way to compute the trajectory is introduced using Matlab. The trajectories have rotation described only around the z axis and this makes the trajectories somewhat odd but since the leaf fall is afterthought in our mind it is not that noticeable. Next a way how to represent these trajectories efficiently using low dimensional structure D on GPU is presented. Then the simulation computation is described using this structure D. Since we render the leaves as simple quads we have a halo artefact around the leaves, this can be fixed if we used a better geometry.

The last chapter before this discuses how fast this implementation is and what are possible ways how to improve performance. This is done for all the the presented tasks, but a proper measurement is done only for the Fall Simulation. This is because the others are offline methods and the tree simulation is considered incorrect.

As for the improvements and further extensions of this work there is a lot of things that could be done. First the texture acquisition part could be created into a standalone application to create a leaf texture. It would also be desirable to add the high frequency detail to the leaves (micro vein and higher detail). Another thing that could be done is creation of a proper geometry for the leaves instead of just a quad. For the Tree Simulation chapter as already mention a further study into the matter would be necessary and make it correct. And of course a lot of optimization could be done. The implementation is a basic one and there is a lot of possible way how to speed up the algorithm.

References

[1] Xiaomin Wang, Chunjiang Zhao, Shenglian Lu, and Xinyu Guo. *Survey on modeling and visualization of plant leaf color.* In: *2009 Third International Symposium on Plant Growth Modeling, Simulation, Visualization and Applications*. 2009. 417–424.

[2] Daeyeoul Kim, and Jinmo Kim. Procedural modeling and visualization of multiple leaves. *Multimedia Systems*. 2017, 23 (4), 435–449.

[3] Yinling Qian, Jian Shi, Hanqiu Sun, Lei Ma, Yanyun Chen, Qiong Wang, and Pheng-Ann Heng. Layered leaf texturing using structure-guided model. *Graphical Models*. 2019, 103 101029.

[4] SoHyeon Jeong, Si-Hyung Park, and Chang-Hun Kim. *Simulation of morphology changes in drying leaves.* In: *Computer Graphics Forum*. 2013. 204–215.

[5] Siyuan Zhu, and Meil Wang. *Plant leaves visualization based on leaf vein extraction.* In: *Proceedings of the Seventh International Symposium of Chinese CHI*. 2019. 101–104.

[6] Ning Zhou, Weiming Dong, and Xing Mei. *Realistic simulation of seasonal variant maples.* In: *2006 Second International Symposium on Plant Growth Modeling and Applications*. 2006. 295–301.

[7] Ying Tang, Dong-Yan Wu, and Jing Fan. Computational approach to seasonal changes of living leaves. *Computational and Mathematical Methods in Medicine*. 2013, 2013

[8] Xiaoming Wei, Ye Zhao, Zhe Fan, Wei Li, Suzanne Yoakum-Stover, and Arie Kaufman. *Blowing in the wind.* In: *Symposium on Computer Animation*. 2003. 75–85.

[9] Chengyang Li, Jingye Qian, Ruofeng Tong, Jian Chang, and Jianjun Zhang. GPU based real-time simulation of massive falling leaves. *Computational Visual Media*. 2015, 1 (4), 351–358.

[10] Brett Desbenoit, Eric Galin, Samir Akkouche, and Jerome Grosjean. *Modeling Autumn Sceneries..* In: *Eurographics (Short Presentations)*. 2006. 107–110.

[11] Ed Quigley, Yue Yu, Jingwei Huang, Winnie Lin, and Ronald Fedkiw. Real-time interactive tree animation. *IEEE transactions on visualization and computer graphics*. 2017, 24 (5), 1717–1727.

[12] KDE. *Krita*.
`https://krita.org/en/`.

[13] Mahes Visvalingam, and J. Duncan Whyatt. The Douglas-Peucker Algorithm for Line Simplification: Re-evaluation through Visualization.. *Comput. Graph. Forum*. 1990 , 9 (3), 213-228.

[14] Sean Barret. *STB*.
`https://github.com/nothings/stb`.

[15] Jin-Mo Kim. Contour-based procedural modeling of leaf venation patterns. *Journal of Korea Game Society*. 2014, 14 (5), 97–106.

[16] David H Douglas, and Thomas K Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: the international journal for geographic information and geovisualization.* 1973, 10 (2), 112–122.

[17] Khalid Khan, DK Lobiyal, and Adem Kilicman. A de Casteljau Algorithm for Bernstein type Polynomials based on (p, q)-integers. *arXiv preprint arXiv:1507.04110.* 2015,

[18] Giang H Dao. *Bresenham/DDA line draw circuitry.* 1996. US Patent 5,570,463.

[19] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools.* 2000,

[20] Serge Beucher, and Christian Lantuéjoul. *Use of Watersheds in Contour Detection.* workshop published. 1979 .
`http://cmm.ensmp.fr/~beucher/publi/watershed.pdf`.

[21] Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Süsstrunk. *SLIC Superpixels* . 2010.

[22] David J. Heeger, and James R. Bergen. *Pyramid-Based Texture Analysis/Synthesis.* In: *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques.* New York, NY, USA: Association for Computing Machinery, 1995. 229–238. ISBN 0897917014.
`https://doi.org/10.1145/218380.218446`.

[23] Ken Perlin. *Improving Noise.* In: *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques.* New York, NY, USA: Association for Computing Machinery, 2002. 681–682. ISBN 1581135211.
`https://doi.org/10.1145/566570.566636`.

[24] Auburn - Jordan Peck. *FastNoise Lite.*
`https://github.com/Auburn/FastNoiseLite`. 2021.

[25] Kim Kulling. *Open-Asset-Importer-Lib.*
`https://www.assimp.org`.

[26] Joey de Vries. *Learn OpenGL.*
`https://learnopengl.com`.

[27] Blender Online Community. *Blender - a 3D modelling and rendering package.* 2018.
`http://www.blender.org`.

[28] Jeremy Berchtold. Pomegranate: Procedural 3D Tree Creation via User-Defined L-systems. 2021,

[29] Nimish Oliapuram, and Subodh Kumar. Realtime forest animation in wind. 2010, DOI 10.1145/1924559.1924586.

[30] Edward Quigley. *Tree Animation and Modeling Via Analytic Simulation and Image-based Reconstruction.* Stanford University, 2019.

[31] *Density of wood in kg/m3, g/cm3, LB/FT3 – The Ultimate Guide.*
`https://matmatch.com/learn/property/density-of-wood`.

[32] Tatsumi Sakaguchi, and Jun Ohya. *Modeling and Animation of Botanical Trees for Interactive Virtual Environments.* In: *Proceedings of the ACM Symposium on Virtual Reality Software and Technology.* New York, NY, USA: Association for Computing Machinery, 1999. 139–146. ISBN 1581131410.
`https://doi.org/10.1145/323663.323685`.

[33] Haoran Xie, and Kazunori Miyata. Real-time simulation of lightweight rigid bodies. *The Visual Computer*. 2014, 30 DOI 10.1007/s00371-013-0783-7.

[34] Cunbiao Lee, Zhuang Su, Hongjie Zhong, Shiyi Chen, Mingde Zhou, and Jie-Zhi Wu. Experimental investigation of freely falling thin disks. Part 2. Transition of three-dimensional motion from zigzag to spiral. *Journal of Fluid Mechanics*. 2013, 732 DOI 10.1017/jfm.2013.390.

[35] Yoshihiro Tanabe, and Kunihiko Kaneko. Behavior of a Falling Paper. *Physical review letters*. 1994, 73 1372-1375. DOI 10.1103/PhysRevLett.73.1372.

[36] Yoshihiro Tanabe, and Kunihiko Kaneko. Behavior of a Falling Paper. *Physical review letters*. 1994, 73 1372-1375. DOI 10.1103/PhysRevLett.73.1372.

[37] Ocornut. *Dear ImGui*.
`https://github.com/ocornut/imgui`.