UNIVERSITY OF MINNESOTA

This is to certify that I have examined this copy of a Master's thesis

By

Umesh J. Maitipe

and have found that it is compete and satisfactory in all respects,

and that any and all revisions required by the final

examining committee have been made.

Dr. Peter Willemsen

Name of Faculty Advisor

Signature of Faculty Advisor

Date

GRADUATE SCHOOL

USING PROGRAMMBLE GRAPHICS HARDWARE FOR REAL TIME TREE ANIMATION WITH SIMULATED WIND PATTERNS

A Thesis

Submitted to the faculty

of

University of Minnesota, Duluth

by

Umesh J. Maitipe

In partial fulfillment of the requirements for the degree

of

Master of Computer Science

July 20

© Umesh Janaka Maitipe 2007

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank Dr. Willemsen, my faculty advisor, for the guidance given to me throughout the thesis work. His guidance was especially valuable since, the work I was doing with programmable graphics hardware involved recent developments in hardware technology, which made it very difficult to get any help from external sources such as, the internet.

I would also like to thank my thesis committee members Dr. Stech and Dr. Dunham for their time and the valuable advice given to me.

I am also grateful for the support given to me by the staff of the Computer Science Department and by my colleagues.

ABSTRACT

Maitipe, Umesh J. M.S., University of Minnesota Duluth, July 2007. Using Programmable Graphics Hardware for Real Time Tree Animation with Simulated Wind Patterns. Major Professor: Dr. Peter Willemsen.

Tree animation plays a vital role in generating realistic natural environments. Due to the massive amount of calculations needed for generating such animation, current prevailing methods are forced to balance computational efficiency against computational complexity. A method which models all the features of a tree's leaf and branch movement will involve a higher degree of computational complexity and thereby will have to sacrifice computational efficiency.

The proposed solution features the 3D movement of branches and leaves while maintaining efficiency of a real time system by porting part of the calculations into the graphics processing unit (GPU). This ability to port calculations into the GPU was made possible due to recent developments which enabled programming the GPU. This transition of calculations has enabled generating a system with realistic tree movement in real time. The movement of the trees in the environment will be governed by the wind patterns present in that environment.

TABLE OF CONTENTS

Page

LIST OF TABLES ………………………………………………………………….. iii

LIST OF FIGURES …………………….…………………………………………. iiiiii

1 Introduction ………………………………………………………………………. 1

2 Generating a Tree Model ………………………………………………………. 4

2.1 Popular Approaches used in generating tree models ……………………….. 4

2.2 Our Approach ……………………………………………………………….. 4

2.2.1 Use of random numbers in generating the tree model ……………..... 8

2.3 Skinning …………………………………………………………………….. 9

2.3.1 Skinning in the Tree model ……..…………………………………… 10

2.3.2 Calculating eight vertices around each existing branch vertex ……... 10

2.3.3 Using the vertices in the `vertices` array to complete the skinning … 12

2.3.4 Use of Vertex Buffer Objects ….…………………………………… 14

2.3.5 Manipulating the vertices in the `vertices` array before storing in a VBO ……………………………………………………………… 15

2.4 Leaves in the tree model …………………………………………………... 16

2.4.1 Display Lists and why one it preferable here over an VBO …………... 17

3 ANIMATING THE TREE MODEL ……………………………………………… 19

3.1 Branch Motion ……………………………………………………………... 19

3.1.1 Resulting force of the wind ………………………………………… 19

3.1.2 Position of the branch in the tree …………………………………… 24

3.1.3 Movement of other branches connected to the one in consideration .. 25

3.1.4 Tree type…………………………………………………………….... 31

3.2 Leaf Motion ………………………………………………………………. 31

3.2.1 Achieving randomness through the use of “noise” ………………… 32

3.2.2 Freedom of movement of a leaf and use of noise to realize it ……….. 32

4 ACCESSING AND ANALYZING THE ENVIRONMENTAL WIND FIELD ….. 35
4.1 Nature of the wind field in the virtual environment …………………………. 36
4.2 Accessing and storing the wind field data …………………………………… 37
5 PROGRAMMING THE GPU TO INCREASE EFFICIENCY …………………… 41
5.1 GPU and its programmability ……………………………………………….. 41
5.2 Calculations that were moved to the GPU to increase efficiency …………… 43
6. RESULTS ……………………………………………………………………… 46
6.1 Implementing the system without utilizing special features of the GPU …….. 47
6.2 Stage 1: Use the high performance memory of the GPU to store data ………. 48
6.3 Stage 2: Transfer calculations to the GPU while utilizing its memory ………. 49
6.4 Adding skinning to trees ……………………………………………………… 50
6.5 Adding leaves to the trees …………………………………………………… 51
7. FUTURE WORK ……………………………………………………………… 53
8. REFERENCES ……………………………………………………………….... 55
A. ALGORITHM TO CALCULATE BRANCH ROTATIONAL ANGLES ……….. 59
B. FRAGMENT SHADER FOR BRANCH ROTATIONAL ANGLE CALCULATIONS ……………………………………………………………..... 60
C. ALGORITHM TO CALCULATE ROTATIONAL ANGLES FOR LEAVES …… 71
D. FRAGMENT SHADER FOR LEAF ROTATIONAL ANGLE CALCULATIONS 73

LIST OF TABLES

Table Page

6.1 Performance of the system without utilizing special GPU features ……………. 47

6.2 Performance when utilizing the GPU high performance memory ………......... 48

6.3 Using the High performance memory, vertex processor and fragment processor of the GPU ………………………………………………………………. 49

6.4 Performance when skinning is added to the trees ………………………………. 50

6.5 Performance when leaves are added ……………………………………………. 51

LIST OF FIGURES

Figure Page

2.1 Final tree model 5

2.2 Application of L-system for tree modeling 6

2.3 Needed Information to calculate the end vertices of the sub-branches 7

2.4 Using random numbers to create realistic looking branches 8

2.5 Applying skinning to the tree model 10

2.6 Calculating the vertices necessary to do skinning 12

2.7 Applying skinning on the tree skeleton 13

2.8 Using triangles to form the surface 13

2.9 Storing vertices in a VBO without manipulation 15

2.10 Storing vertices in a VBO after manipulation 16

2.11View of a leaf and its initial orientation 17

2.12 Leaf rotated around the Z-axis and transferred to a new position 18

3.1 Result of collective application of `angleXY` and `angleYZ` 20

3.2 Tree branch attributes 22

3.3 (a) Wind is blowing at time t1 in same direction as at time t0 and stronger ... 23

3.2 (b) Wind is blowing at time t1 in same direction as at time t0 but weaker or if the wind had changed directions 23

3.4 keeping track of the branch levels 25

3.5 With no communication between different levels of connected branches 25

3.6 With proper communication between different levels of connected branches .. 26

3.7 Rotating the branch around the Z-axis by 30 degrees 26

3.8 Translating the branch to a new point 27

3.9 Finding the final coordinates of the point at (x2, y2, z2) after applying a rotation around the Z-axis and a transformation 28

3.10 Use of glPushMatrix and glPopMatrix commands 29

3.11 Freedom of movement of a leaf around the X and Z-axis …………………… 33
3.12 Rotational motion around the petiole axis …………………………………… 34
4.1 Wind Field Information ……………………………………………………… 35
4.2 Nature of the wind field ……………………………………………………….. 36
4.3 Horizontal and vertical wind spans …………………………………………….. 37
4.4 Wind field measurements ……………………………………………………. 38
4.5 Sequence in which branches are accessed ……………………………………… 39
4.6 Correct mapping of the wind vectors …………………………………………… 40
5.1 Using the GPU to calculate the new rotational angles for branches and leaves … 45
6.1 Summarizing the results in Tables 6.1, 6.2 and 6.3 ……………………………… 50
6.2 Multiple trees forming a forest ……………………………………………....... 52

1. INTRODUCTION

Tree animation, whether it exists in a scene of a film or in the background of a game scene, will significantly effect how realistic looking the scene will be. The more realistic it is, the greater the satisfaction of the interacting person. Due to the importance that realistic tree animation has on an environmental scene, many approaches have been followed, resulting in various degrees of success. Approaches that have been adopted towards building a successful model can be categorized as being non-real-time, or real-time.

Non-real-time systems are suitable for situations where the quality of the image generated is significantly important to the scene and the time it takes for the algorithm is of lesser importance. Such is the case when developing an animation film. Image Based Rendering (IBR) is an approach used on such occasions [1]. IBR refers to generating extremely high-quality images based on those captured by cameras when observing a real scene [1]. Apart from this not being a real-time system, other problems (such as discontinuities in the created model due to occlusions in the original image) have indicated the need for alternative methods [2].

In real time systems the quality of the images is sacrificed to various extents in order to maintain the computational efficiency of the system. For instance, trees can be represented using planar billboards, which involve projecting a 2D view of an object (e.g. [3]). Such an approach is simple to implement as it will only involve a 2D representation, and therefore it consumes very little computing resources [3]. However trees so generated will lack any form of reality at close range and animations of such a model will lack complexity. Another approach involves producing tree movement based on random calculations rather than basing it on qualities such as spring constants of branches and the wind the trees are subjected to. This form of simplification will reduce the amount of calculations that need to be done to determine the appropriate wind loads affecting the tree branches and then determining the appropriate swaying angles of the

branch and leaves. Such a system will perform appropriately when considered as an isolated unit. However when introduced into an environment where other units (e.g. hair of people, dust particles, clothes) are recognizing and moving according to the existing wind patterns, such random movement will be inadequate.

The goal of our approach is to animate in real time both branches and leaves of several 3D trees according to the wind patterns in the environment. Approaches that have pursued a similar goal are mentioned in [4] and [5]. Both these approaches have successfully implemented the following features of our goal:

- Animating both leaves and branches of a 3D tree
- Animation is done in real time
- Animation is not random but is done in accordance to external forces. However, these forces do not have any relevance to environmental wind forces.

In order to completely realize our goal the following three additional features were specifically incorporated into our method:

- Include the ability to detect the changes in the wind field and adjust the animation accordingly
- Reduce the amount of calculations handled by the central processing unit (CPU) by passing part of the calculations to the graphics processing unit (GPU) using OpenGL Shading Language (GLSL)
- Render the movements of leaves and branches of multiple trees instead of rendering the movements of a single tree.

The reduction of the workload of the CPU has enabled us to animate leaves and branches of multiple trees in a real, time while at the same time doing the necessary complex calculations needed to integrate the wind field changes into the animation.

In the following sections, we will first look into the approach followed in generating the tree model, and then we will consider how the movement of the tree branches and leaves are calculated so that they are consistent with both the external forces acting on them as well as their connection to the rest of the tree. Next, we will explore in-depth how the environmental wind field is mapped to affect branch movement in the trees. We will also look into the important role played by GLSL in minimizing the workload of the CPU. Finally, we will look at a set of results which show the performance gain we have experienced as we move calculations into the graphics card, while at the same time adding more features to the system.

2. GENERATING A TREE MODEL

2.1 Popular Approaches used in generating tree models

Procedural models are quite popular for modeling trees [6]. L-systems in particular have being used extensively for this purpose [7, 8, and 9]. An L-system is a recursively defined algorithm which has clearly defined rules governing each recursive step. By increasing the number of recursive steps it is possible to generate realistic looking complex tree models. However due to the underlying algorithm that governs an L-system it is very difficult to generate trees with distinct shapes. This lack of functionality has resulted in other methods. One such method involves extracting information from photographs of a particular tree and then coupling that information with an L-system to produce the model of that tree [7]. Another alternative method proposes using captured images of a tree to define the volume data of a tree and constructing the branch structure using simple branching rules [2].

2.2 Our Approach

We have opted for an L-system approach while clearly defining the rules that govern the recursive steps to suit our needs. An example of one of the trees produces using our approach is shown in Fig 2.1. There will be two variables that govern the complexity and nature of a tree, and these variables can be adjusted by the user on a tree by tree basis before rendering the scene.

`iter` : This variable will specify the number of recursive calls (Fig 2.2).

`numD`: This variable will specify the number of sub-branches each branch will get divided into (Fig 2.2).

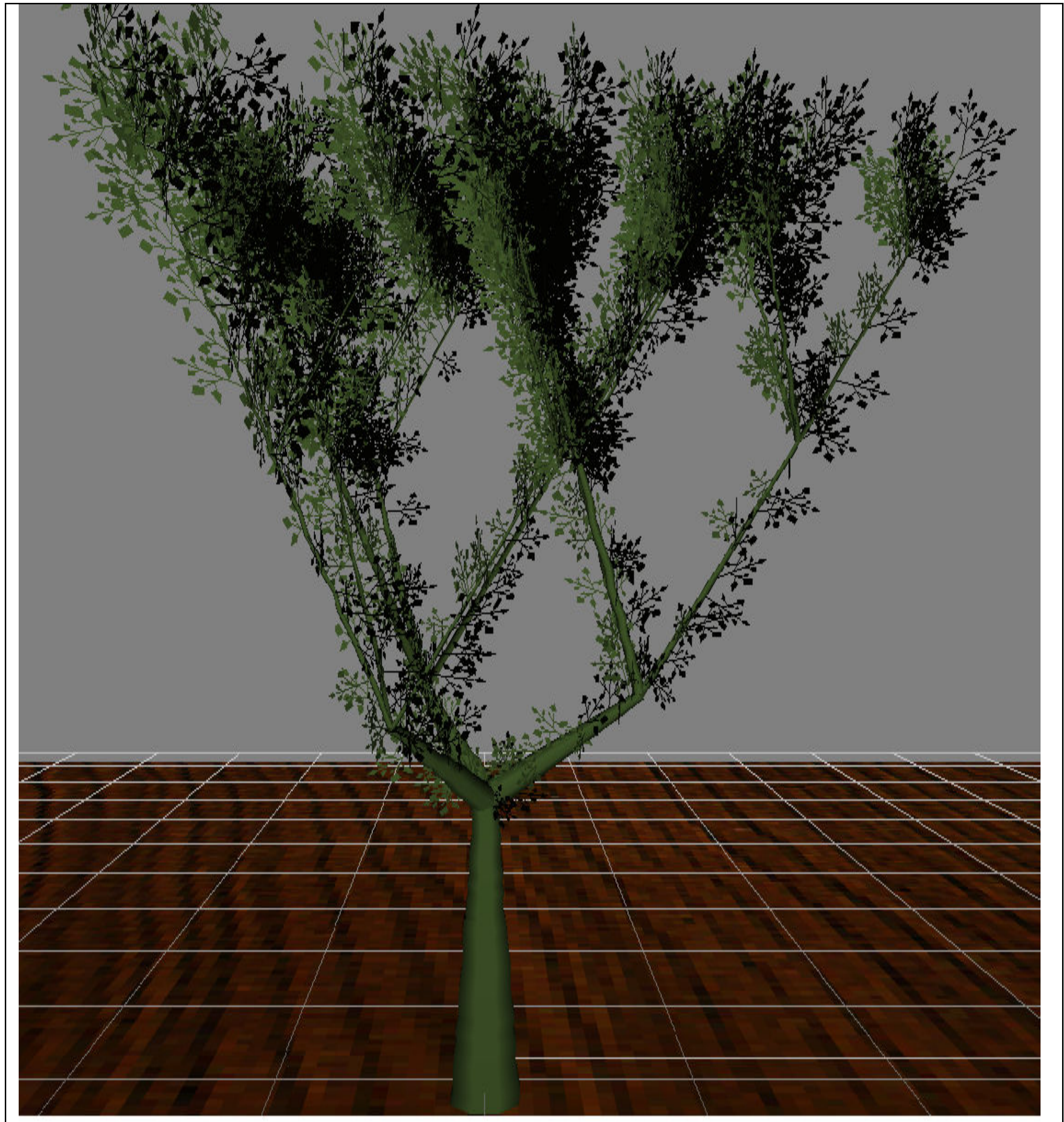
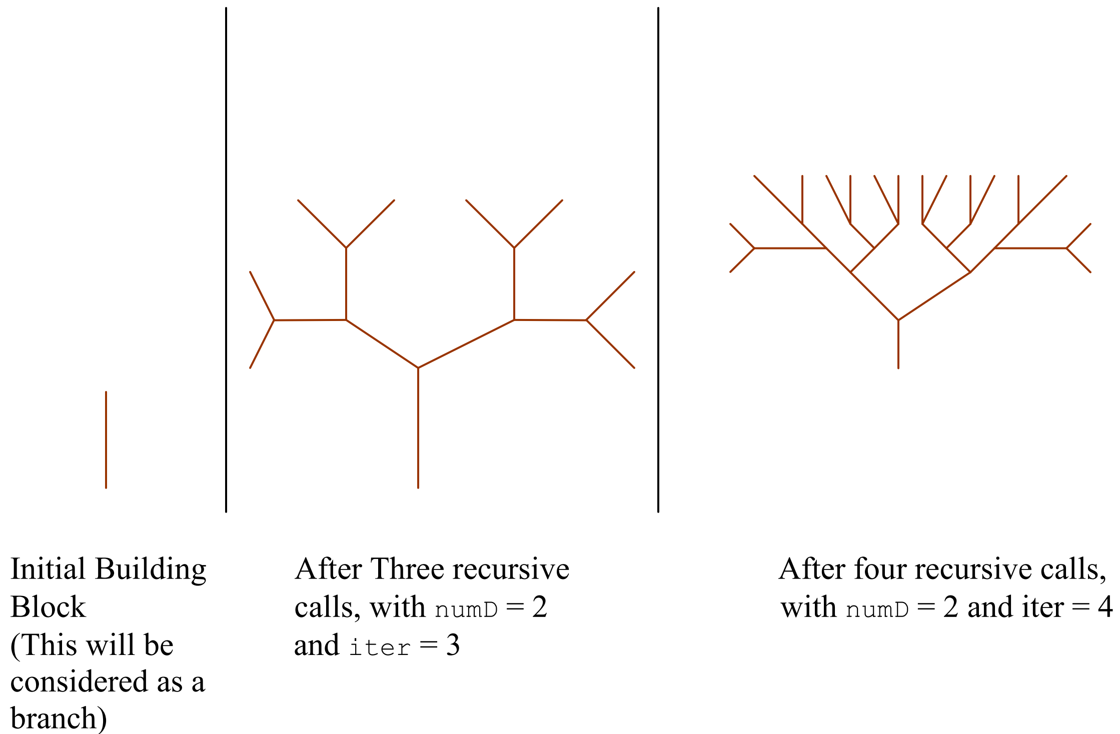


Fig 2.1 Final tree model

Fig 2.2 shows how the model will change as we vary the above mentioned two variables. It is important to keep track of the vertices that define the branch structure of the tree. This is because it is these vertices that are manipulated in order to animate the tree.



Initial Building Block (This will be considered as a branch)

After Three recursive calls, with `numD` = 2 and `iter` = 3

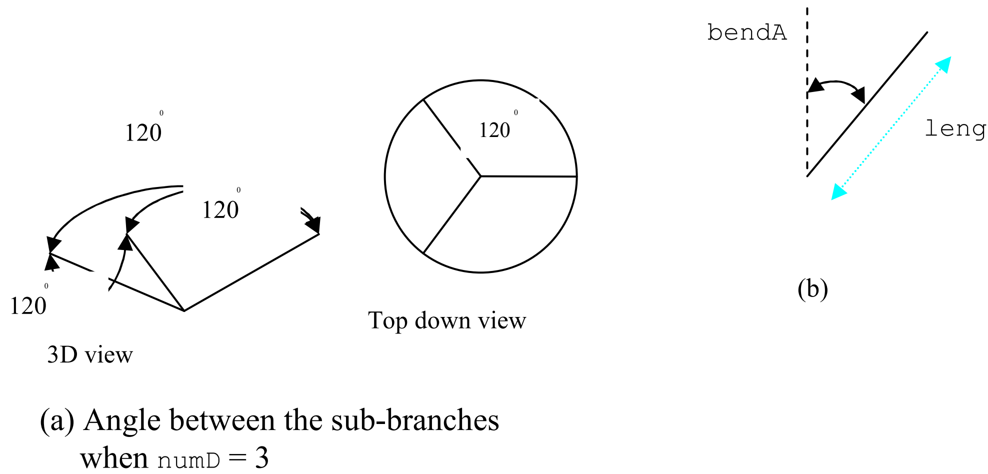
After four recursive calls, with `numD` = 2 and iter = 4

Fig 2.2 Application of L-system for tree modeling

Since the branch structure is defined recursively, the calculations that are done to find out the new vertices when a branch is divided into sub-branches also need to be done recursively. The goal of keeping track of vertices can be converted into a situation where we have to calculate the new end vertices equivalent to `numD` at each subdividing point of a branch. The information we need for this is as follows:

1. The value of `numD`
2. The angle between the new sub-branches (Fig 2.3 (a))
3. The length of a branch (`leng` Fig 2.3 (b))
4. The angle between the branch and the vertical axis (`bendA` Fig 2.3 (b))

5. The end vertex of the main branch from which new branches will divide out.



(a) Angle between the sub-branches when `numD` = 3

Fig 2.3 Needed Information to calculate the end vertices of the sub-branches

Given this information, we are able to determine the new endpoints of the sub-branches while taking the endpoint of the parent branch as the starting vertex of all sub-branches. All these start and end vertices of branches will be stored in an array known as `verticesForLeaves` with the intention of its use in successive calculations. Without any other modifications, if we were to use the same predetermined set of rules in our recursive steps, the resulting branch structure will have a very uniform appearance. It is possible to affect this appearance to some extent by introducing random values in our calculations. However, due to the large number of calculations that need to be done to animate the trees, any component added with the intention of improving the appearance of the trees might create considerable negative effects on performance. This is because the new additional component might involve calculations which will reduce the rate at which the system completes the calculations, thereby making the system not operational in real-time. Therefore, we will not diverge drastically from the uniform nature of our calculations. This will result in maintaining a simple realistic looking structure while animating the tree model with less processing power.

2.2.1 Use of random numbers in generating the tree model

After the `verticesForLeaves` array has been properly defined and populated we have all the information to draw the skeleton of a tree with straight lines. However it should be noted that tree branches are naturally not straight but are rather bent in many places. We will use random numbers in our tree model to create a more realistic branch appearance.

Once the start and end vertices of a branch are established, that information will be sent into a function whose main purpose will be to insert a certain number of other vertices in-between the start and end vertices. By introducing these new vertices, it is expected to define the branch as a collection of sub-branches, resulting in a much more realistic appearance (Fig 2.4).

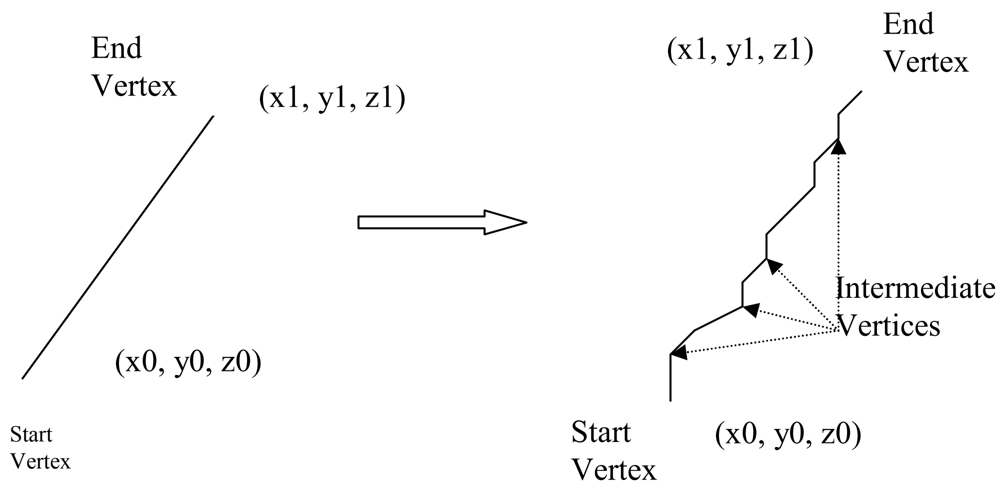


Fig 2.4 Using random numbers to create realistic looking branches

The number of vertices that will be used in breaking a branch into pieces will be defined before rendering starts (this variable is known as `numBreaks`). The functionality of this function can be expressed by the following steps:

1. Determine the differences in the x, y and z coordinates of the start and end vertices.
    a. end.x – start.x
    b. end.y – start.y
    c. end.z – start.z
2. Next, divide these differences by the `numBreaks` so that it can be determined by how much we have to increment the x, y, and z coordinates of the start vertex to get to the next vertex.
3. Add a random value when determining the next vertex coordinates so that the end result will not likely be a straight line.

2.3 Skinning

At this point, the skeleton of the tree is generated. However we have to use a process know as skinning to get the realistic appearance of a tree. Skinning refers to the process of coating a skeleton of a model with an appropriate surface representation. There are two forms of skinning:

- Smooth skinning: Here, each point in the surface that is generated will be connected with more than one underlying skeleton vertex. Also, the deformation of a surface point with regard to the deformations of the skeleton will be based on how much weight is assigned to the connections between the surface point and the skeleton points it is connected to [10].
- Rigid skinning: Here, each surface point is only connected with one skeleton point [10].

The type of skinning used in our model is rigid skinning. This is because tree bark will have a rigid texture, therefore surface points need not be manipulated by the movement of more than one skeleton point. On the other hand, when representing human skin, it is important to use smooth skinning to obtain a realistic look.

2.3.1 Skinning in the Tree model

The approach used for generating a surface representation can be briefly described as follows: around each branch vertex in the `verticesForLeaves` array we establish eight vertices, as shown in Fig 2.5. We store these new vertices in the `vertices` array and use these vertices to draw the branches. This will complete the skinning process. Now, let's look at each of these steps in more detail.

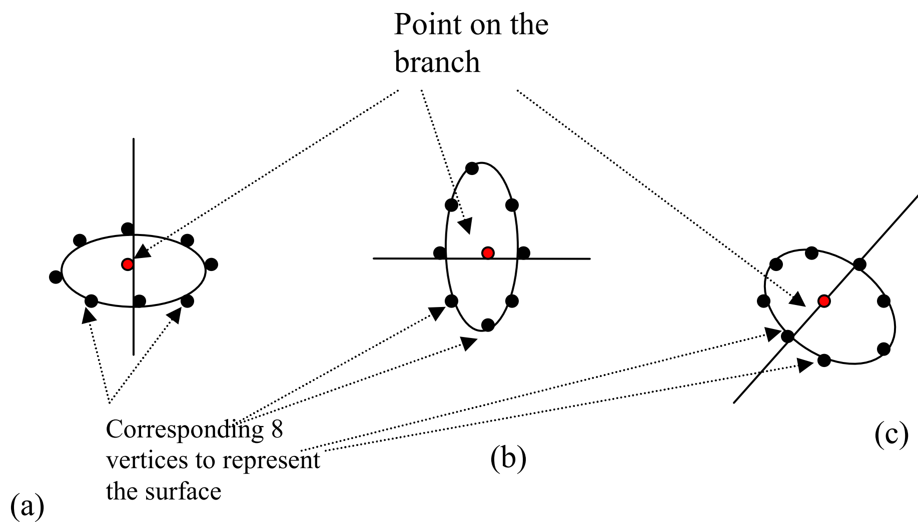


Fig 2.5 Applying skinning to the tree model

2.3.2 Calculating eight vertices around each existing branch vertex

The reason I decided to use eight vertices is mainly as an act of balancing appearance against computational complexity. If I had used four vertices, the resulting branch would have a box like appearance, while six would have produced an effect quite pleasing to the eye, but still generated a branch that lacks the roundness that is associated with tree branches. Eight vertices (and anything more than that) did produce the necessary roundness, but as we increase the number of vertices that will make up the surface, it will also increase the number of vertices that need to be manipulated when

animating the trees. Therefore it was decided to replace each skeleton branch vertex will only eight vertices when applying skinning.

As shown in fig 2.5 (a), (b), and (c) the branch can have different orientations. Therefore the process of calculating the eight vertices around a given vertex has to take into account the orientation of the branch which can change from tree to tree. Also, given the fact that establishing the branch vertices is a recursive exercise, it would be more appropriate if this new functionality was also recursively carried out. Considering these factors, the steps below were followed in establishing the surface vertices.

Needed information:

a) The thickness of the branch both at the starting point and at the end point.
b) The value of `numD`
c) The vertex around which we are establishing the eight points.
d) The angle between the branch and the vertical axis

Once the above information is passed, the calculations can be carried out as follows:

1. For each vertex using the information mentioned above, calculate all the possible different orientation vertices as shown in Fig 2.6. The reason for making a general calculation of this nature is to ensure that when this functionality is used recursively, it is able to handle any vertex in the tree.

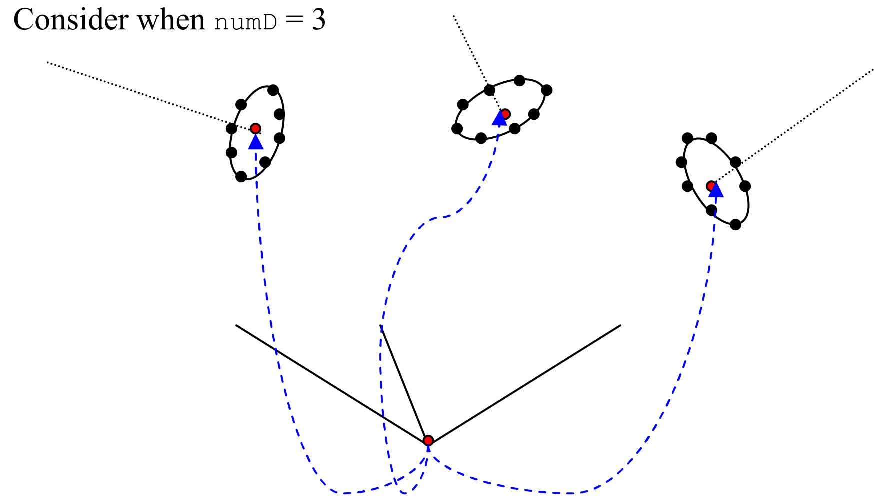
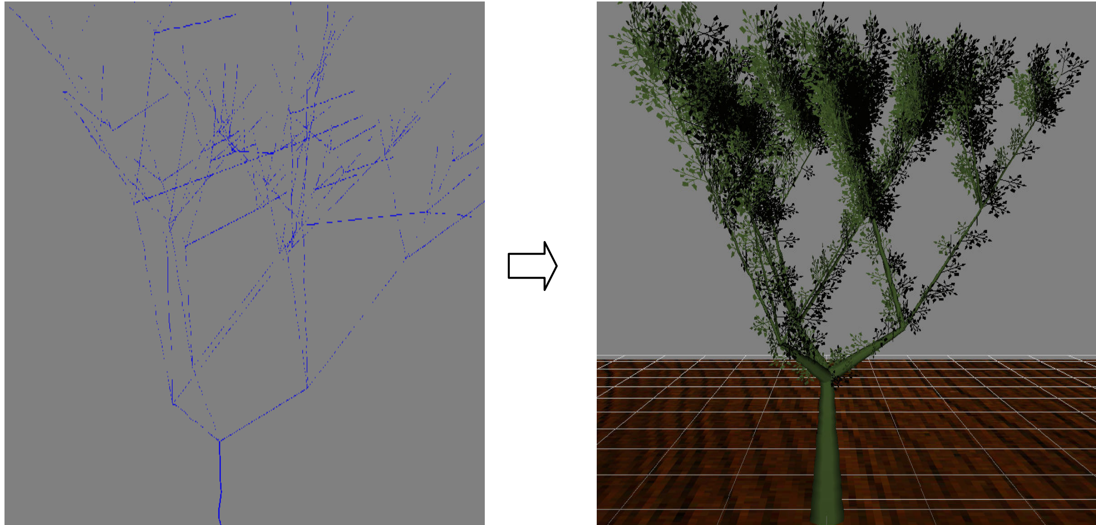


Fig 2.6 Calculating the vertices necessary to do skinning (We consider the branches separately, as it is difficult to separately identify the new surface vertices if they are drawn together.)

2. Once the above set of vertices is calculated, it is important to select the set of vertices that corresponds to the orientation of the branch that is currently being considered. This calculation is simplified by our assumption that all branches are sub-divided into the same number of sub-branches.
3. Each representing set of surface vertices is stored in the `vertices` array.

2.3.3 Using the vertices in the `vertices` array to complete the skinning

Once the vertices needed to represent the surface of the tree are calculated and stored in the `vertices` array, the question of connecting them in a suitable manner to construct a realistic looking tree arises (Fig 2.7).



Before Skinning After Skinning

Fig 2.7 Applying skinning on the tree skeleton

This can be accomplished by allowing the surface vertices to form triangles as shown in Fig 2.8. Since the final rendering process of the tree model will be more efficient if the vertices are stored as Vertex Buffer Objects (VBO), the vertices in the `vertices` array will be further manipulated.

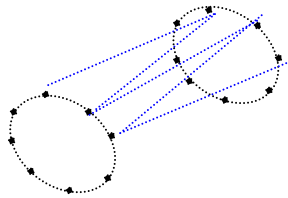


Fig 2.8 Using triangles to form the surface

2.3.4 Use of Vertex Buffer Objects

VBO are a powerful feature as they enable rendering data to be stored in the high performance memory of the graphics card. This means calculations that employ the data will be executed more quickly than if the data were stored in the CPU because the time to transport the data for processing will be reduced. In our case the data will be the `vertices` array.

When using vertices stored in a VBO to draw geometric objects, two situations can arise:

a) The vertices of the object are stored to match the sequence in which OpenGL commands access them for rendering and therefore it is possible to access a complete chunk of information from the VBO and carryout the rendering.
b) The vertices are not stored in the correct sequence. If this is the case it is not possible to directly access a complete block of information as it is. Then using an index array with the VBO becomes necessary. The index array is responsible for indicating the sequence in which the vertices should be accessed in order to draw the geometric shape in question.

Storing of vertices in the `vertices` array falls into the second category. However, there can be a problem in creating an index array to accompany the `vertices` array because the number of vertices is dynamic and therefore it is not possible to proclaim a fixed order in which vertices should be stored. The reason the number of vertices is dynamic is because the number of branches that a tree can have varies from tree to tree. In this situation, what can be done is to manipulate the vertices so that they are in the proper sequence before storing them in the VBO. Then an index array is not necessary, and it is possible to access continuous blocks of memory from the VBO during rendering.

2.3.5 Manipulating the vertices in the `vertices` array before storing in a VBO

Sets of eight vertices (representing the outer layer of the tree skeleton) are currently stored in the `vertices` array (Fig 2.9). However, in order to use these vertices to draw the necessary triangles that make up the outer layer, they will be manipulated and stored in a new array called `newArr` (Fig 2.10). Once this new array is created, it is possible to store that array information into a VBO and then efficiently render the tree model at rendering time.

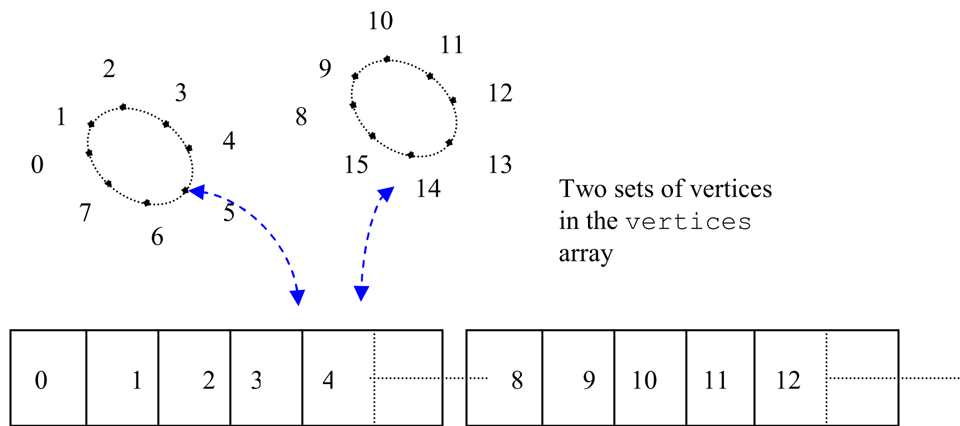
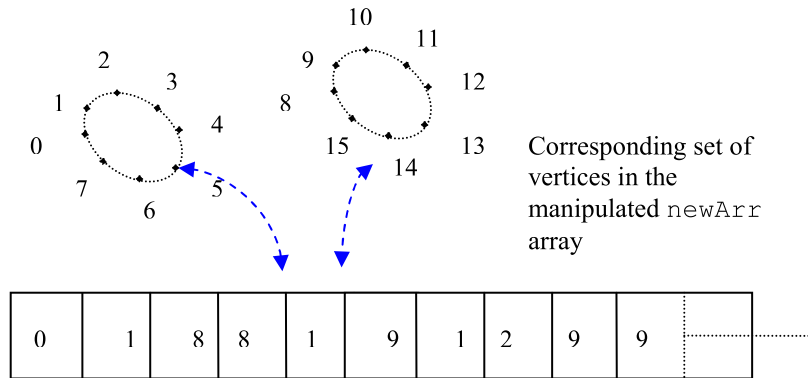


Fig 2.9 Storing vertices in a VBO without manipulation



Sequence of vertices appear as above in the VBO

Fig 2.10 Storing vertices in a VBO after manipulation

The basic idea behind manipulating the vertices in the `vertices` array is that even though the number of branches will change, a branch will always have a constant number of breaks (`numBreaks`). In our program, this value is fixed at 16. This means that the surface of a branch will consist of 16 sets and each set will have eight vertices. Given the consistency in the number of vertices defining a branch, it is possible to treat it as a unit and define how the vertices in such a unit should be ordered in order to get an array similar to that explained in Fig 2.10.

2.4 Leaves in the tree model

Leaves are an important factor in any tree model. A tree can contain hundreds of leaves however, it is important to note that each and every leaf need not have a unique feature distinguishing them. Actually, it is more natural for leaves to have the same appearance. With these ideas in mind, we have defined a single leaf as shown in Fig 2.11. Unlike branches where VBOs were utilized, in order to increase the efficiency of the rendering process, for leaves we will use a single display list (as explained below) to store the particulars of the leaf unit.

This whole cluster will be treated as
a single element representing a leaf.

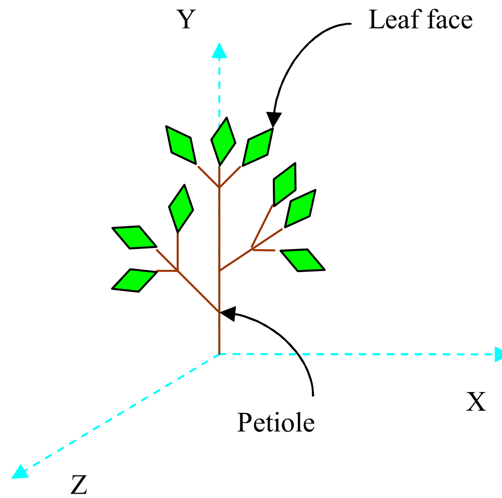


Fig 2.11 View of a leaf and its initial orientation

2.4.1 Display Lists and why one is preferable here over an VBO

A display list is a simple group of OpenGL commands. Such a list can be used to define an image. A display list should be used when such a group of commands needs to be executed a large number of times. In such a case, instead of executing that particular set of commands from the CPU memory, all the necessary information can be stored in the high performance cache memory of the graphics card. This will ensure that the rendering will get done efficiently. It should be noted that once a display list is created by defining a set of commands it is not possible to alter it.

The reason for using a display list over a VBO is because in this situation we need not calculate the vertices of the leaves and manipulate them. Because (as mentioned above) it is acceptable for leaves to have a similar appearance, a leaf will be constructed as shown in Fig 2.11 and the specification of this construct will be stored within a display

list. Then, we will rotate and translate the initial leaf rendered by the display list in order to represent other leaves. This process is further explained below:

a) Establish the display list to draw a leaf with its petiole along the Y-axis and its face perpendicular to the Z-axis. The leaf's starting point will be at the origin of its own local coordinate system.
b) Rotate the leaf around the Z-axis (Fig 2.12)
c) Rotate the leaf around the X-axis
d) Transfer it to its new position (Fig 2.12)

However it should be noted that even though the same display list is used again and again this doesn't mean that we are replicating the same leaf multiple times. This is just a case of using the same shape multiple times while the rotational angles and the points to which the leaves are translated will be stored separately.

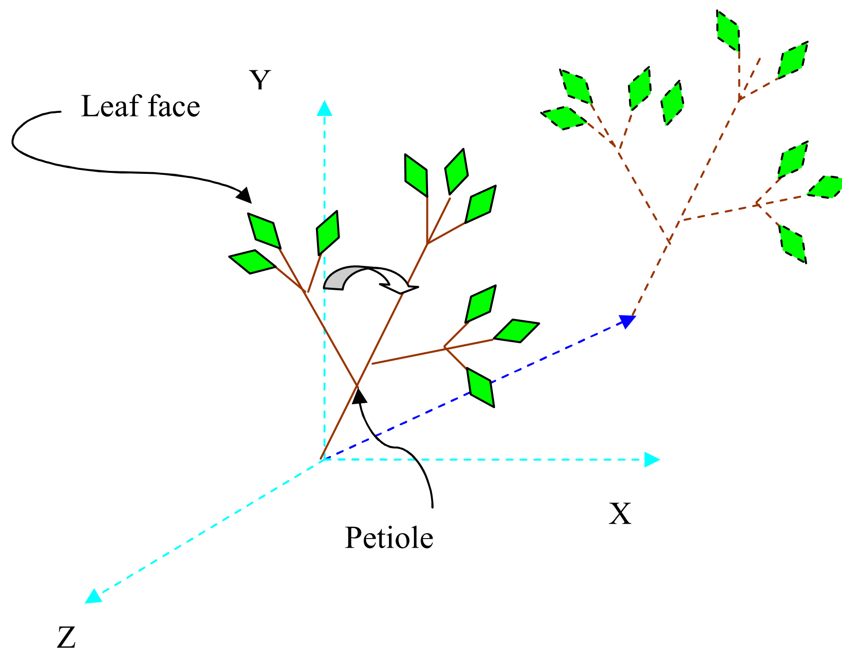


Fig 2.12 Leaf rotated around the Z-axis
and transferred to a new position

3. ANIMATING THE TREE MODEL

There has been a lot of effort to create realistic tree animations ([2, 4, 5, 11, and 12]). The method we employ to animate the leaves and the branches has certain similarities to those employed in the system described in article [4]. The reason for choosing to explore the approach described in [4] over [2, 5, 11, 12] is because it deals with both branch and leaf animation while [2, 5, 11, 12] only focus on branch animation. We did not attempt to diverge considerably from an already proposed method because the changes introduced by us focuses on employing new approaches to carry out calculations more efficiently rather than introducing a novel approach to manipulate tree branches and leaves. Our approach will enable us to deal with a large number of trees instead of just a single tree, which was the case with the system described in [4].

3.1 Branch Motion

There are many potential factors affecting a branch's movement. Out of these the most important and the ones that have the greatest impact on branch motion are:

- Resulting force of the wind
- Position of the branch in the tree
- Movement of other branches connected to the one in consideration
- Tree type

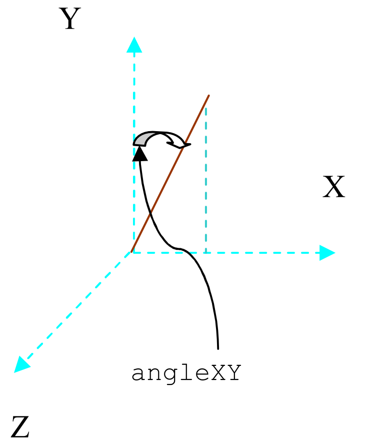
In order to accurately replicate the movement of the branches we will take into account each of the above factors in our calculations.

3.1.1 Resulting force of the wind

This is the main factor that will affect the branch movement. There are two variables belonging to each branch that will be modified based on the wind force: `angleXY` and `angleYZ`.

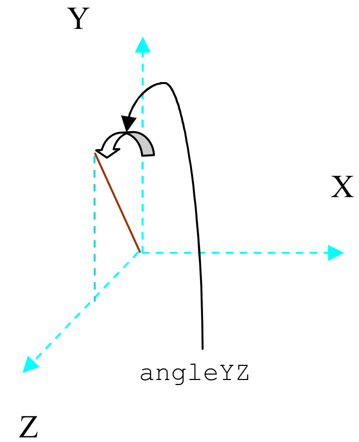
a) The rotation angle in the XY-plane due to the wind : `angleXY` (Fig 3.1 (a))

b) The rotation angle in the YZ-plane due to the wind : `angleYZ` (Fig 3.1 (b))



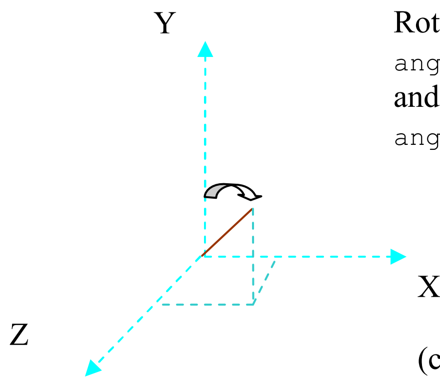
Rotating the branch by `angleXY` around the Z-axis

(a)



Rotating the branch by `angleYZ` around the X-axis

(b)



Rotating the branch by `angleXY` around the Z-axis and then rotating it by `angleYZ` around X-axis

(c)

Fig 3.1 Result of collective application of `angleXY` and `angleYZ`

The calculations necessary in order to find the values of `angleXY` and `angleYZ` will be based on the cantilever flat-spring model [4].

By defining collective movement in two planes the replicating movement we generate will be 3D (Fig 3.1 (c)). It is assumed that the wind vector acting on a branch (also called "wind load") is consistent along the entire length of the branch. Consider the following calculations involved in modifing `angleXY` and `angleYZ`:

1. Find the wind vector at the base of the branch under consideration : `wind`
2. Use that wind vector to calculate the corresponding forces acting on the branch.

    a) current wind load acting along the x-axis : `newXLoad = wind.x`
    b) current wind load acting along the z-axis : `newZLoad =  wind.z`

    c) load exerted on the branch due to `newXLoad :  xBranchLoad`
    d) load exerted on the branch due to `newZLoad :  zBranchLoad`

    e) `xBranchLoad = newXLoad + maxBranchLoad * n1`
    f) `zBranchLoad = newZLoad  + maxBranchLoad * n2`

    g) `n1` and `n2` are random values in the range [-1, 1]. The `maxBranchLoad` is a non-directional term that will affect the directional loads `newXLoad` and `newXLoad` [4]. By adding a non-directional term into the calculation it is expected that the internal forces acting at the joint of the branch will be compensated.

3. Now that the exact forces acting on the branch are known the next step is to find out the resulting deformations due to these forces.

    Deflection along the X-axis due to `xBranchLoad : defX`

Deflection along the Z-axis due to `zBranchLoad : defZ`

```
defX = xBranchLoad / k
defZ = zBranchLoad / k
```

`k` is defined as the spring constant of the timber (Fig. 3.2). This is a variable defined as follows [4]:

k = {elastic modulus * width of the branch * (thickness) ^ 3} / {4 * (span length) ^ 3}

Elastic modulus is a unique value for each tree. However, the other terms used in the calculation will differ from branch to branch. This will ensure that the behavior of branches due to the wind will not be uniform.

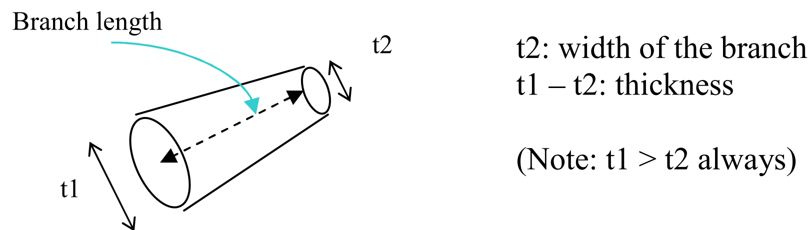


Fig 3.2 Tree branch attributes

4. Find out the angles by which the branch should be rotated around X and Z axis to match the deflection calculated in step 3.

   Rotating angle around the X-axis: `motionAngleX`
   Rotating angle around the Z-axis: `motionAngleZ`

   `motionAngleX` = arcsin(`defX` / branch length)
   `motionAngleZ` = arcsin(`defZ` / branch length)

5. Calculate the final rotational angles (Fig 3.3)

`angleXY` at time t1 = `angleXY` at time t0 + `motionAngleX`

`angleYZ` at time t1 = `angleYZ` at time t0 + `motionAngleZ`

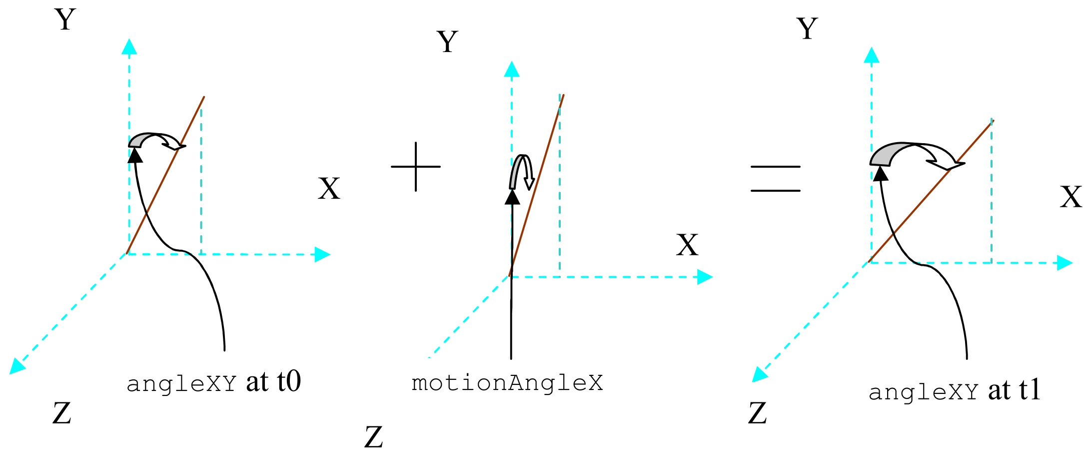


Fig 3.3 (a) Wind is blowing at time t1 in same direction as at time t0 and stronger

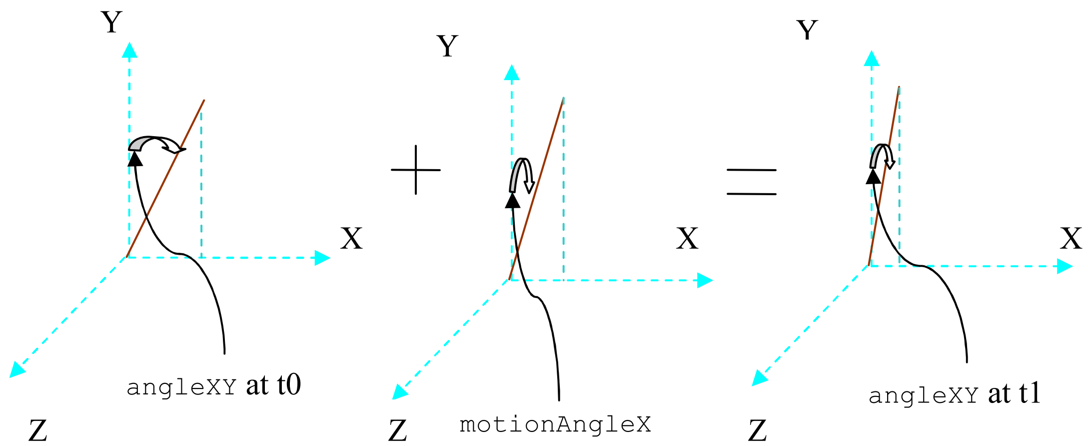


Fig 3.3 (b) Wind is blowing at time t1 in same direction as at time t0 but weaker or if the wind had changed directions

At the end of the above calculation it should be checked that `motionAngleX` and `motionAngleZ` do not exceed the maximum angle that the branch can rotate in any given direction (`maxAngle`). If any of those angles are greater than the `maxAngle` then they will be given the value of the `maxAngle`.

Once the final values of `angleXY` and `angleYZ` are calculated, if the branch is rotated by the values of these new angles directly it will result in movements which are very fast and very much unlike the smooth motion that is associated with branch motion. In order to get the characteristic smooth motion each angle change will be divided into a large number of partial increments or decrements and applied accordingly.

3.1.2 Position of the branch in the tree

Position of the branch in the tree is also an important factor. For instance branches at the top most level will normally show much more movement than those at a lower level. Some of this difference in activity can be attributed to the thickness of the branches, which has being accounted for during the spring constant calculation where thickness is included in the calculations. However, some of this movement is due to other reasons, such as the higher levels of the trees having more leaves and therefore more influenced by the wind. Apart from this, higher branches are younger than those at lower levels and even if they have the same thickness the resistance at the joints against movement is less.

We will keep track of the level of a branch using a variable called `branchLevel` (Fig 3.4) The effect of `branchLevel` will be expressed by multiplying the wind load by a factor that is a function of the `branchLevel`. This will ensure that higher branches will move faster even when the wind field is uniform along the vertical axis.

```
windLoad = windLoad + factor * branchLevel
```

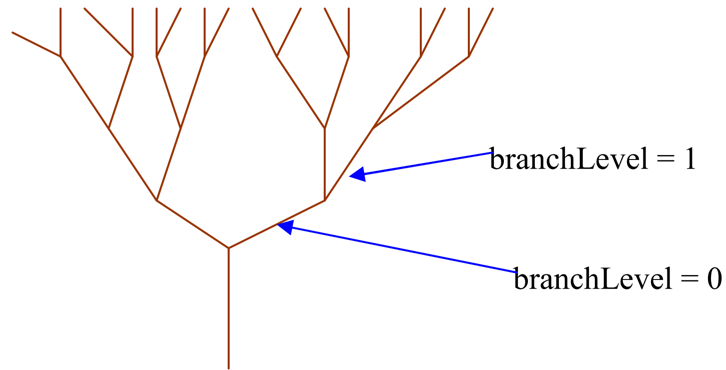


Fig 3.4 keeping track of the branch levels

3.1.3 Movement of other branches connected to the one in consideration

Each branch will be considered as a single unit with its own local coordinate system. However, if the branches were allowed to move independently without any form of communication between them this will result in flying branches, where branches will not be connected to any joint (Fig 3.5).

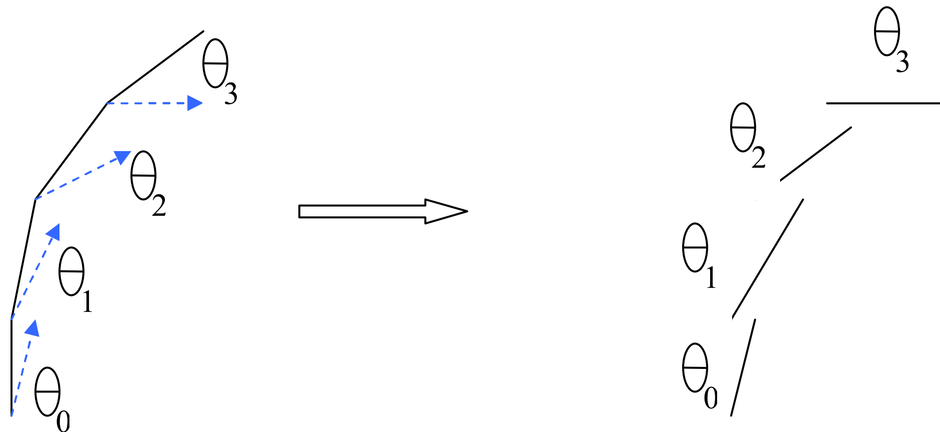


Fig 3.5 With no communication between different levels of connected branches

Therefore, there should be some method through which the branch at a top level gets to know about the movement of all the connected branches below it (Fig 3.6).

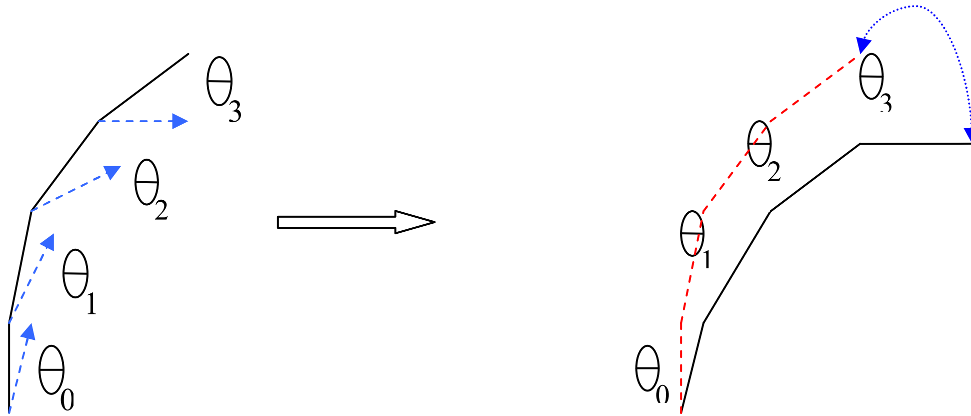
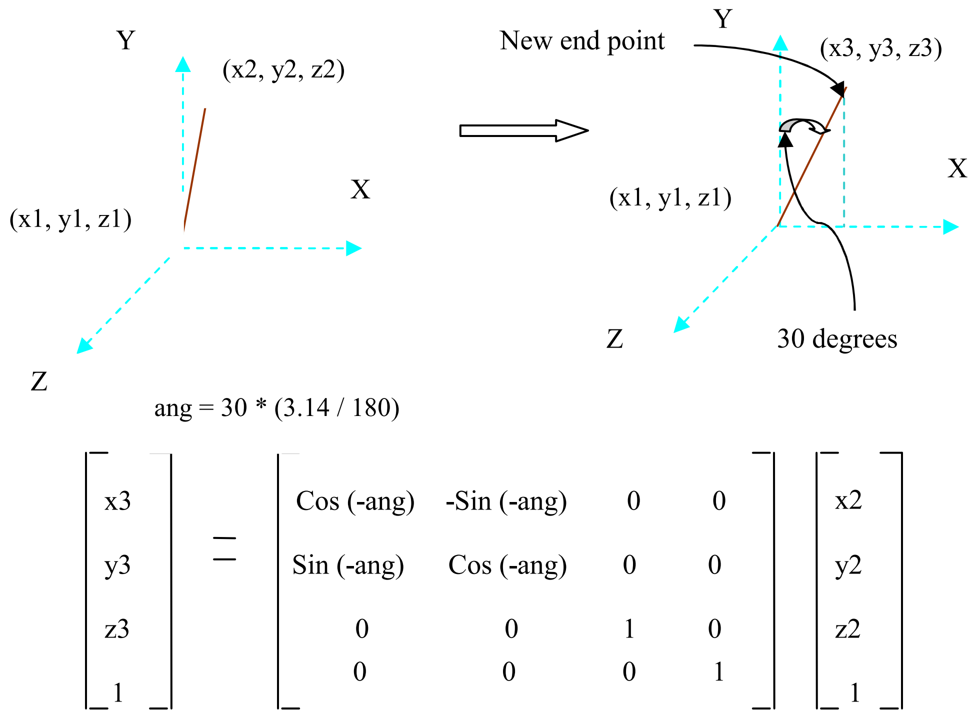


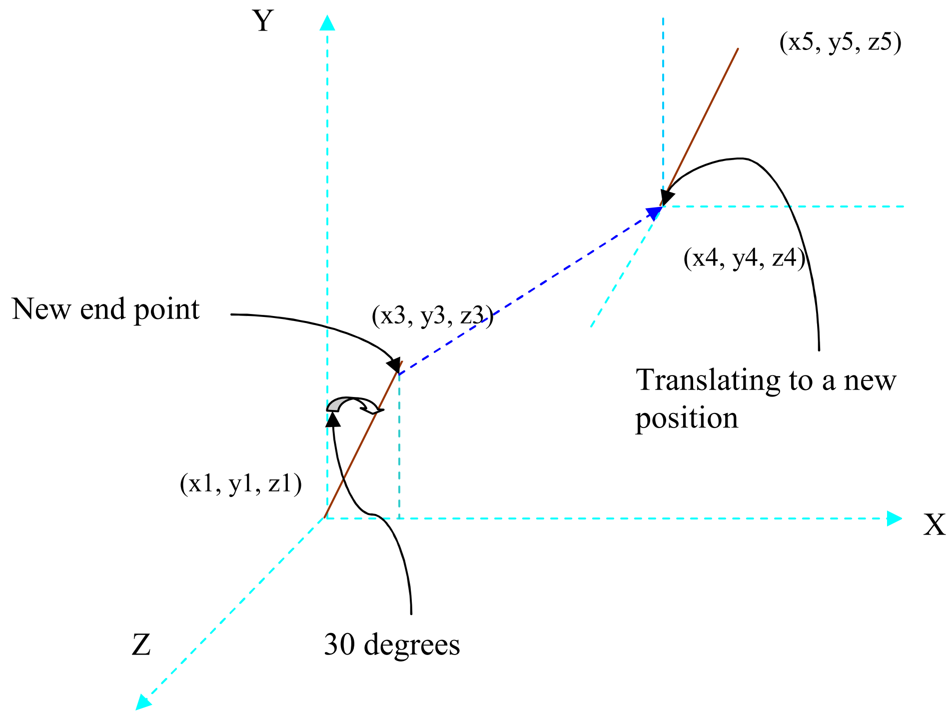
Fig 3.6 With proper communication between different levels of connected branches

Each transformation of the branch will involve a matrix multiplication. For instance consider the situation of rotating a branch by 30 degrees around the Z-axis (Fig 3.7) and then translating it to a new point (Fig 3.8).



$$\begin{bmatrix} x3 \\ y3 \\ z3 \\ 1 \end{bmatrix} = \begin{bmatrix} \text{Cos (-ang)} & \text{-Sin (-ang)} & 0 & 0 \\ \text{Sin (-ang)} & \text{Cos (-ang)} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x2 \\ y2 \\ z2 \\ 1 \end{bmatrix}$$

Fig 3.7 Rotating the branch around the Z-axis by 30 degrees

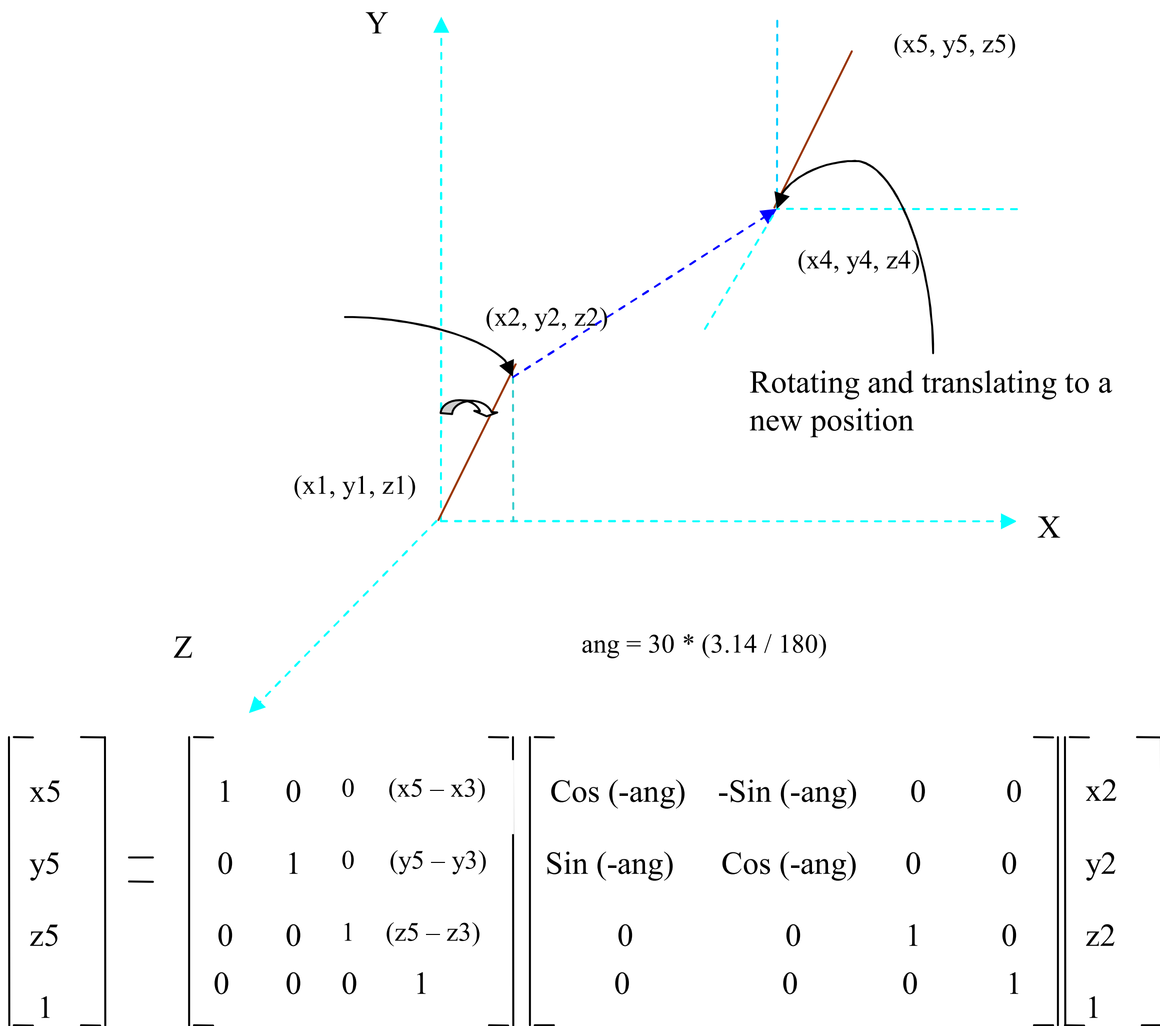


$$\begin{bmatrix} x4 \\ y4 \\ z4 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & (x4 - x1) \\ 0 & 1 & 0 & (y4 - y1) \\ 0 & 0 & 1 & (z4 - z1) \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x1 \\ y1 \\ z1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x5 \\ y5 \\ z5 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & (x5 - x3) \\ 0 & 1 & 0 & (y5 - y3) \\ 0 & 0 & 1 & (z5 - z3) \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x3 \\ y3 \\ z3 \\ 1 \end{bmatrix}$$

Fig 3.8 Translating the branch to a new point

In Fig 3.7 and Fig 3.8 each transformation is considered as an isolated incident and the matrices that will generate the new coordinates are shown. However, it is possible to combine these transformations together and the resulting coordinates of the final transformation can be obtained by multiplying the matrices of individual transformations (Fig 3.9).



$$\begin{bmatrix} x5 \\ y5 \\ z5 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & (x5 - x3) \\ 0 & 1 & 0 & (y5 - y3) \\ 0 & 0 & 1 & (z5 - z3) \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \text{Cos (-ang)} & \text{-Sin (-ang)} & 0 & 0 \\ \text{Sin (-ang)} & \text{Cos (-ang)} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x2 \\ y2 \\ z2 \\ 1 \end{bmatrix}$$

Fig 3.9 Finding the final coordinates of the point at (x2, y2, z2) after applying a rotation around the Z-axis and a transformation.

It is important to note at this point that OpenGL has two commands that enable keeping track of these transformation matrices. Those two commands are glPushMatrix and glPopMatrix. It is possible to use these two commands to keep track of transformations that are connected to each other and isolate those that do not have any connection.

glPushMatrix: Transformation matrices are maintained in a stack. The glPushMatrix command will create a copy of the top most matrix in the stack and insert it to the top of the stack. This will ensure that we have a saved point in the line of transformation to which we can return. For example, if matrix `m1` was multiplied by two matrices `m2` and `m3` resulting in a new matrix `m3`, then the stack will contain matrix `m1` and on top of that matrix `m3`. By using the glPushMatrix command at this point, it can be ensured that a copy of `m3` resides on the top of the stack.

glPopMatrix: This command will return the matrix that is one below the top of the stack. This will basically get us back to the point we saved using the glPushMatrix command.

Consider Fig 3.10: branches numbered 1, 2, and 3 are interconnected while branches numbered 1, 4, and 5 are also interconnected as a separate group. However, branches 2 and 3 need not know about the movement of branches 4 and 5.

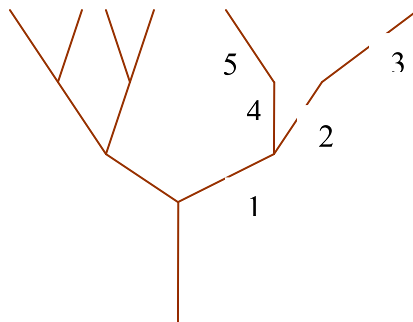


Fig 3.10 Use of glPushMatrix and glPopMatrix commands

The drawing of these two sets of branches incorporating the relevant transformations will take the following form:

- ❖ Use glPushMatrix to save the current cumulative transformation matrix m1 on the top of the stack. (Save m1)
  - Do the transformation for branch 1 (new matrix m2 on the top)
  - Draw branch 1
  - Use glPushMatrix to save the current cumulative transformation matrix (save m2)
    - Do the necessary transformations for branch 2
    - Draw branch 2
    - Do the necessary transformations for branch 3

      When we are doing the transformation for branch 3, those matrices will get multiplied by the transformation matrices of branch 2. This will ensure that branch 3 will reflect branch 2's transformation also
    - Draw branch 3
  - Do a glPopMatrix (now m2 is back on the top)
  - Use glPushMatrix to save the current cumulative transformation matrix (save m2 again)
    - Do the necessary transformations for branch 4
    - Draw branch 4
    - Do the necessary transformations for branch 5
    - Draw branch 5

      Note that branch 5 will know about the transformation of branch 1 and branch 4 but will not be effected by the transformations of branches 2 and 3.
  - Do a glPopMatrix (again m2 is back on the top)

- ❖ Do a final glPopmatix as there aren't anymore linked branches. (Now the matrix on the top is m1.)

In this manner it is possible to propagate the transformations along a set of linked branches while isolating those transformations from any unconnected branches.

3.1.4 Tree type

Depending on the type of the tree, the movement of the branches will change. This is due to the fact that the internal resistance against movement at the joints will be different from tree to tree. Currently, this factor is being incorporated into the calculations through the spring constant. The calculations for the spring constant involve the elastic modulus, which is a value that changes from tree to tree.

3.2 Leaf Motion

Animation of a tree would not be complete if the motion of tree leaves were not taken into account. Some observations regarding leaves that established the guidelines for generating the motion of tree leaves are as follows:

- Leaves on the whole flutter in a manner that is independent from the direction of the wind.
- Leaves have more freedom of movement than branches
- The motion of two leaves (even if connected to the same branch and positioned nearby each other) need not have any similarity.

After taking the above observations into considerations and also the content of article [4] it was decided that we need not connect the leaf motion directly with the wind field vectors. Therefore, we decided to use the concept of noise to get the desired leaf motion as this would generate the random but smooth movement associated with leaves.

3.2.1 Achieving randomness through the use of “noise”

Noise has quite a few definitions and many uses in the computer field. However, in our case, noise was used with the idea that it will generate a set of numbers which has a pseudo-random appearance. Unlike normal random values, noise values do not have large fluctuations between subsequent values, therefore using such values will ensure that we will obtain the intended smooth leaf movement. The article [4] mentions the use of 1/(f^β) noise based values in animating the leaves. The reason given is that 1/(f^β) noise exists in various natural phenomena, and it is well suited to replicate leaf movements [4]. Currently, we are using Perlin noise instead of 1/(f^β) noise as it is much simpler to create using existing resources [15], and our main intention is to generate the movement while exploring how much of the calculations can be passed to the graphics card to increase the efficiency of the overall system.

3.2.2 Freedom of movement of a leaf and use of noise to realize it

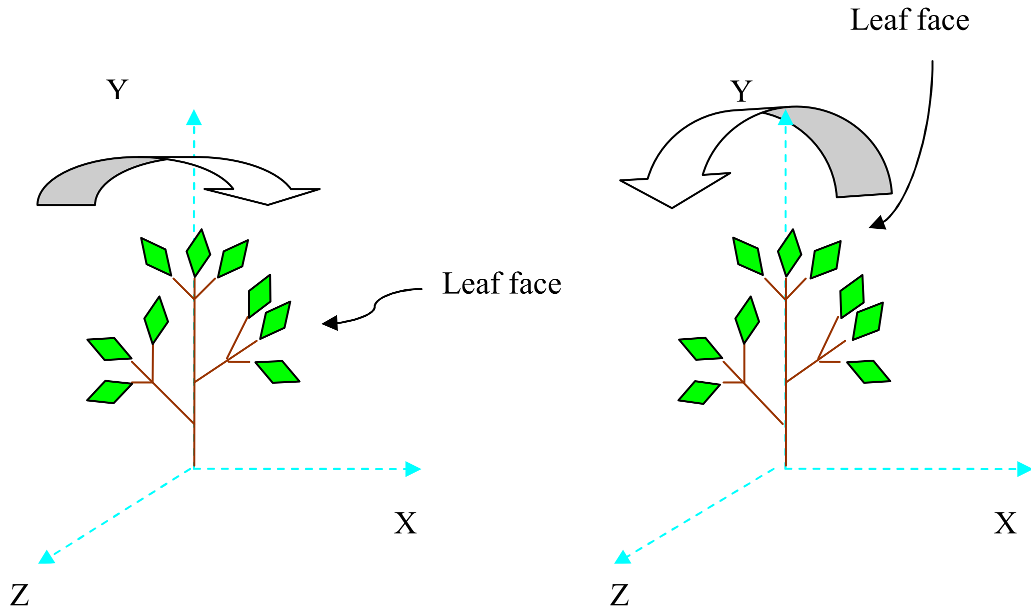
Similar to the movements of a branch, the leaves are also able to move in two planes, as shown in Fig 3.11. The two variables that will represent these movements are:

`motionAngleX` : represent the leaf’s rotation around the Z-axis
`motionAngleZ` : represent the leaf’s rotation around the X-axis

```
motionAngleX = maxMotionAngleX * n1 * factor
motionAngleZ = maxMotionAngleZ * n2 * factor
```

`maxMotionAngleX` : This is the maximum angle by which the leaf can rotate around the Z-axis.
`maxMotionAngleZ` : This is the maximum angle by which the leaf can rotate around the X-axis
`n1` and `n2` are perlin noise values in the range [-1, 1].

`factor` : A variable that can be adjusted to increase or decrease the speed of the leaf motion.



(a) Freedom of movement of a leaf around the Z-axis.

(b) Freedom of movement of a leaf around the X-axis.

Fig 3.11 Freedom of movement of a leaf around the X and Z-axis.

Apart from this, in order to successfully replicate the natural leaf motion, leaves are also allowed to rotate around the petiole axis (Fig 3.12). This movement is represented by a variable called `totalRotaAngle`. This is calculated as the addition of two terms:

```
totalRotaAngle = rotationAngle + helixAngle
```

`rotationalAngle` is obtained using the noise function:

`rotationalAngle` = maximum rotation angle around the petiole axis * noise value

`helixAngle` this is a rotational angle generated around the petiole axis according to the horizontal motion of the leaf.

```
helixAngle = motionAngleX * linkedFactor
```

`linkedFactor` is a variable that can be adjusted to increase or decrease the speed of the leaf motion [4].

It should be noted that each leaf is considered as a separate entity and therefore each leaf will have its own set of values for the above mentioned variables. Through the application of `motionAngleX`, `motionAngleZ`, and `totalRotaAngle` it is possible to generate realistic looking movement of leaves.

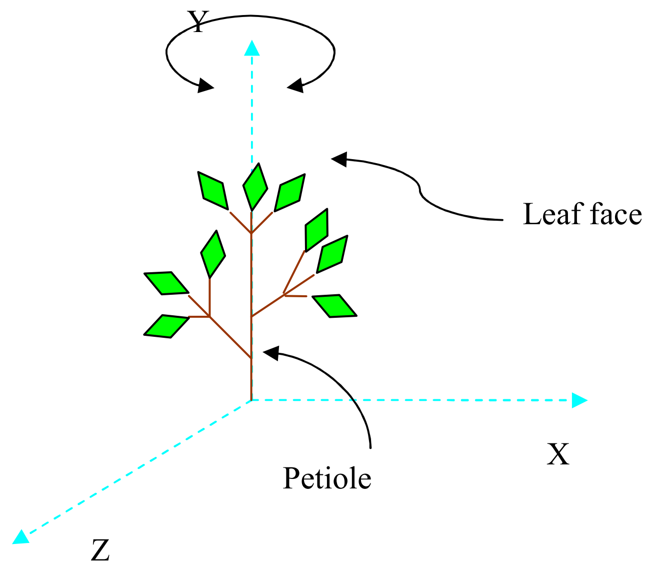


Fig 3.12 Rotational motion around the petiole axis

4. ACCESSING AND ANALYZING THE ENVIRONMENTAL WIND FIELD

Many research efforts [2, 4, 5, 11, and 12] that study the animation of trees consider wind as the main force that animates the branches and leaves. These studies also introduce different equations which enable the determination of the amount of rotations and deflections the branches will be subjected to due to wind forces. However, none of these studies focus on actually accessing the wind field during a given point in time and extracting the exact wind vector in order to carry out the calculations based on that information. The reason for not attempting the above is because a wind field is a 3D domain where every point within the domain will have a distinct wind vector and, as Fig 4.1 suggests, this number of points can easily amount to millions. If the CPU attempts to extract the wind field information and do the necessary computation each time the scene is to be rendered, it will significantly reduce the performance of the system due to the amount of processing needed to be done. However, we will show that due to the ability to pass some of the calculations into the GPU, it is possible to access the wind field and do a proper analysis of the data without a significant loss of performance.

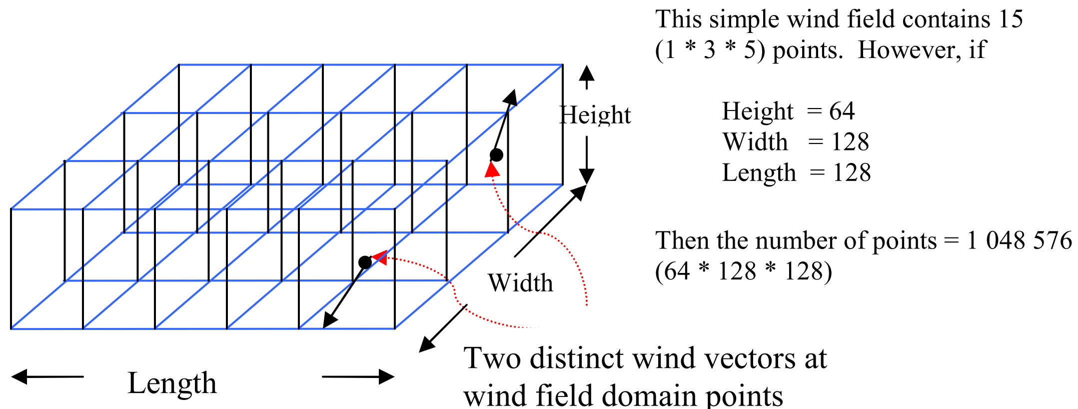


Fig 4.1 Wind Field Information

4.1 Nature of the wind field in the virtual environment

Normally, in a 3D wind field, if wind vectors acting on two points were considered they will be distinct from each other. This will be true in our system also across the horizontal plane. However, we have assumed that, vertically there will not be any changes in the wind vectors (Fig 4.2).

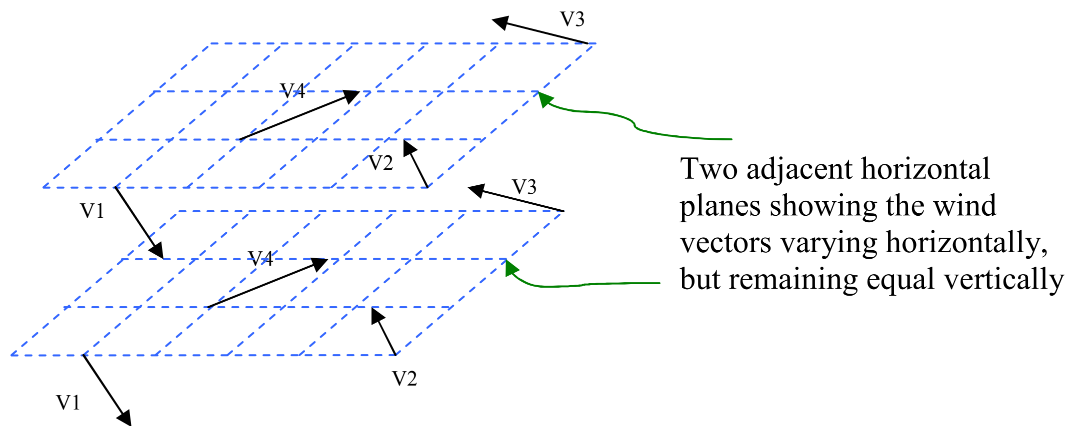


Fig 4.2 Nature of the wind field

This assumtion was made based on the following reasons:

- It is much more difficult to fix the vertical span of the wind field compared with setting the horizontal span (Fig 4.3)
    - The vertical span will change depending on the length of the branches
    - The number of branch levels the trees have
    - Our trees tend to grow more vertically than horizontally, resulting in a large vertical difference for a corresponding small horizontal change.
- Removing the vertical component from the calculations will not result in a significant gain in performance.

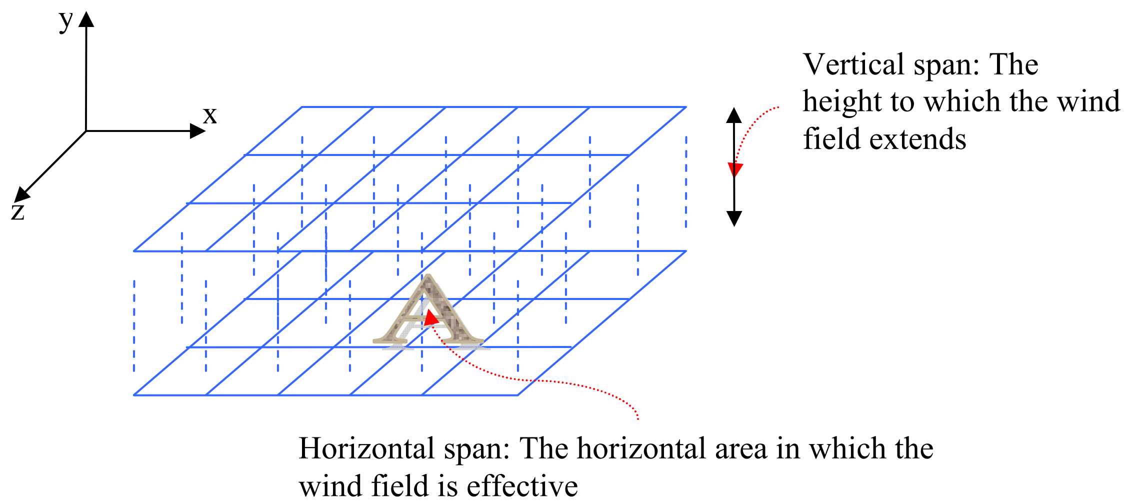


Fig 4.3 Horizontal and vertical wind spans

We have also made a reasonable assumption that there is a system which will provide information on the 3D wind field in real time. However, it is left to us to extract the information needed and map it to suit our calculations. There are many wind simulation systems in existence [13, 18]. By defining the needed operations to extract the needed wind data and map it accordingly, it is ensured that the tree system developed by us can be integrated easily into a system such as in [13], and will move according to the wind patterns defined in that system.

4.2 Accessing and storing the wind field data

Information about the wind field is assumed to be available in the form of a one dimensional array (`forestWind`) of size:

horizontal width * horizontal length * 3 (Fig 4.4)

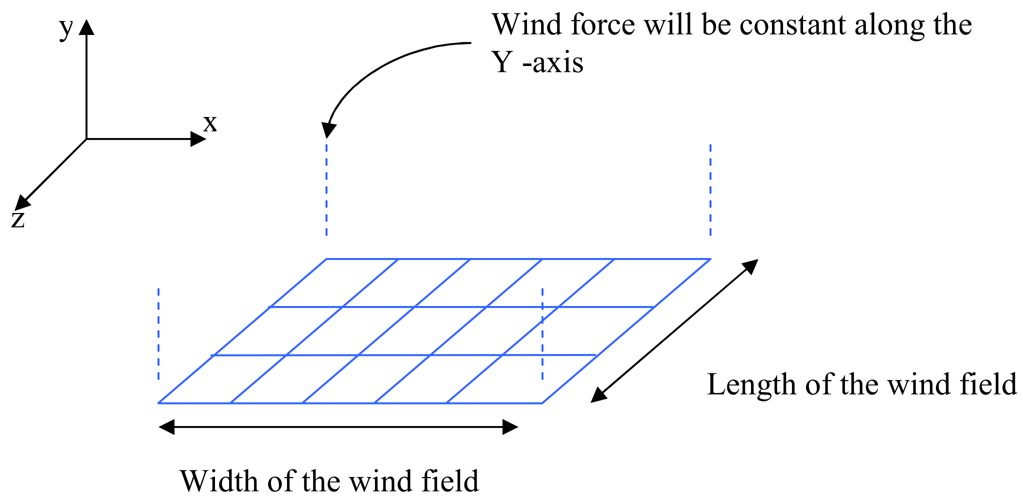


Fig 4.4 Wind field measurements

We will access the wind field at the start point of each branch. In accordance with the reasoning explained in section 3.2, calculating the leaf motion will not involve interaction with the wind field. Therefore, we will narrow our consideration only to branches. If the starting point of a branch is at (x1, y1, z1), then it is possible to access the x, y, and z directional vectors of the wind at (x1, y1, z1) as follows:

`windIndex` = (x1 + z1 * horizontal width) * 3;

wind vector in x direction = forestWind[`windIndex` + 0];

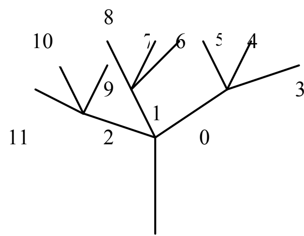
wind vector in x direction = forestWind[`windIndex` + 1];

wind vector in x direction = forestWind[`windIndex` + 2];

This data will be stored in a texture which will be passed to the GPU for additional calculations. However, it is important to note the sequence in which branches of a tree will be accessed, due to the recursive nature in which those calculations are being performed (Fig 4.5). This sequence will be followed in calculating the transformations as

well as drawing the branches. Therefore, the wind information of each branch should also be stored in the same sequence to facilitate proper accessing of data (Fig 4.6).

This creation of a wind texture step will be done each time we render a new scene which means that if the wind changes at any given time it will be reflected in the scene.



(a) Numbering of tree branches

0, 3, 4, 5, 1, 6, 7, 8, 2, 9, 10, 11

(b) Due to the recursive nature of the calculations, this is the order in which the branches will be accessed

Fig 4.5 Sequence in which branches are accessed

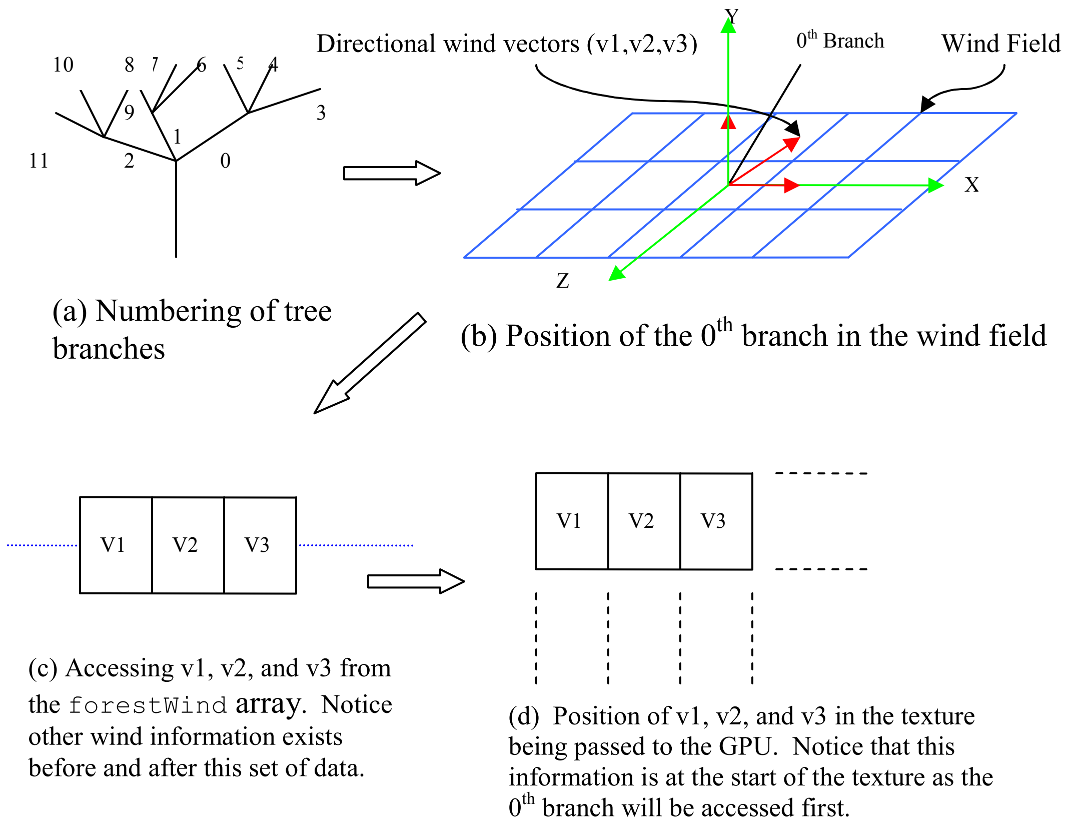


Fig 4.6 Correct mapping of the wind vectors

5. PROGRAMMING THE GPU TO INCREASE EFFICIENCY

5.1 GPU and its programmability

Traditionally the computer only possessed one instruction processing unit which was the CPU. The CPU is responsible for fetching instructions from a secondary storage unit such as the hard disk, storing it within the cache memory, decoding it, and finally executing the instruction, while interacting with the secondary storage units to get any other needed data [24]. However, relying totally on the CPU for processing of instructions and their execution induced a bottleneck in areas such as computer graphics, where calculations such as addition and multiplication of millions of numbers need to be done within a small time period when rendering images. The need to overcome this bottleneck created the demand for a unit whose designated task is to carry out a large number of similar calculations within a short period of time. The created unit with such capabilities is the GPU.

The GPU is a parallel processing unit whose purpose is to render graphics to the screen. Rendering graphics to the screen involves sending graphics data (3D coordinates (x, y, z)) through a set of stages to convert that data into pixels which will be displayed in the 2D space of the monitor. This set of stages forms a model known as the graphics pipeline [14]. The process that takes place within the graphics pipelines can be divided into two main stages [14] and [25]:

- Geometry Processing: Changes 3D object coordinates into 2D window coordinates.
- Pixel Processing: Fill the 2D space of the monitor with pixels to represent the surface of the objects being rendered.

Due to the ability to execute the operations in the graphics pipeline in parallel, the GPU can be considered as a Single Instruction Multiple Data machine (SIMD). This means that the GPU can execute a single instruction such as addition or multiplication on a large number of data simultaneously thereby saving the time needed for calculations. Unlike the CPU the GPU do not interact with the secondary storage units nor does it possess a

cache memory suitable for storing instructions. This is because the instructions that are passed to the GPU from the CPU will directly match algorithms that are built into the GPU, therefore no decoding of instruction is necessary. Also, all the necessary data to carryout those instructions will be passed from the CPU, so the GPU will not need to interact with the secondary storage units in order to fetch data.

Unique features of the GPU include: no expectation of data reuse, independence in manipulating data, therefore the ability to carryout calculations in parallel, and the presence of specifically defined algorithms to carry out a range of calculations such as element-by-element vector addition and multiplication [14, 21, 22 and 23].

It is possible to customize how the graphics pipeline handles the vertex data within the geometry processing stage and calculate the correct color for each pixel on the screen during the pixel processing stage. This ability to customize is known as programming the GPU or programming the graphics pipeline. The first programmable GPU was introduced in 2001 by NVIDIA and it was called GeForce 3. We will use a language called GLSL for these programming purposes. Programs written to customize the geometric processing stage are known as vertex shaders or vertex programs. While those programs used to customize the pixel processing stage are known as fragment shaders or fragment programs. The Vertex shaders will perform vertex and normal transformations and per-vertex lighting computations in order to carryout the duties of the geometry processing stage [25]. The Fragment shaders will function on the following pixel data in order to produce its final red, green, blue, and alpha values [16]:

- Its current red, green, blue, and alpha values
- The pixel position (x and y coordinates) as passed by the vertex program
- The z-coordinate will be used in deciding whether a pixel should be drawn or not, as it contains the depth information for pixels.
- Global information such as position of light sources and the colors of the light sources.

Out of the two programs by which it is possible to specify the operations of the graphics pipeline it is the Fragment Program that we are most interested. This is because our rendering process involves manipulating a large amount of vertex information at a given time. The most efficient way of passing a large amount of information to the GPU is by storing the information in a texture. Currently it is only using a fragment program that we can access information stored in a texture, manipulate it, and write back to a different texture which is then accessible to the CPU. There are different data types defined in GLSL which enables passing data from CPU to GPU [25]. We will be using a data type known as `uniform` for the necessary data passing.

Using a texture for any other purpose other than rendering involves a new perspective. The main factor in this perspective is that each pixel in a texture is made up of four floating point values, and it is possible to manipulate these values within the fragment program. Given this perspective, the next step is storing data that needs to be manipulated using the special features of the GPU in place of these four floating point values. Since it is possible to access the results of the fragment program from the CPU this enables accessing the manipulated data from the texture that was initially passed to the GPU.

5.2 Calculations that were moved to the GPU to increase efficiency

The algorithmic components that get repeated each time the rendering is done are noted below:

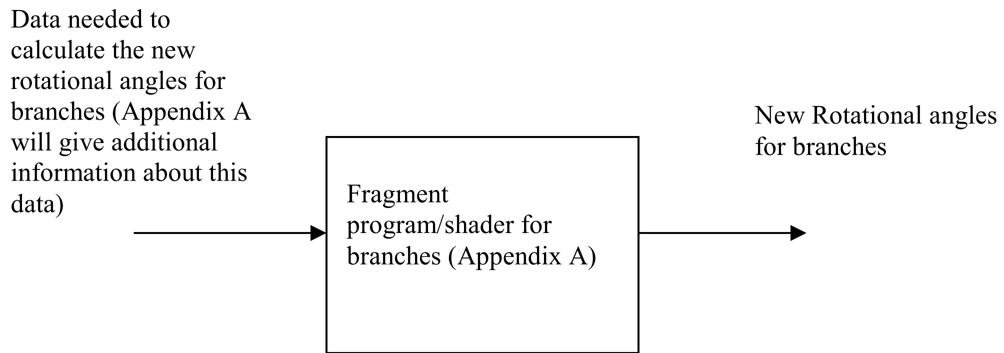
- Calculating the wind field information and entering it into a texture
- Calculation of the angles by which the leaves and branches need to be moved
- Drawing the vertices in their new positions after applying the relevant transformations.

Out of these, calculating the wind field information involves interacting with an external system. Such interaction can be done only by the CPU. Therefore, creating the texture that stores the wind information is handled by the CPU. However, once we have this

data, the next step of calculating the angles by which the leaves and branches need to be moved can be ported to the GPU.

- Using the GPU's ability to do calculations in parallel:
  Calculation of rotational angles does not depend on other branches or leaves. As such, we consider each leaf and branch as a separate entity. Therefore, these particular calculations can be done in parallel.
- Overcoming the GPU's inability to store data for reuse:
  Angle calculation does involve knowing the previous corresponding angles. However when the calculated angles are passed to the CPU that information can be stored within a texture. This enables accessing this texture as part of data when the new angles are to be calculated.
- Using the GPU specific algorithms to mathematically manipulate data:
  The angle calculations involve manipulating floating point values. The GPU does have algorithms for efficiently handling floating point addition and multiplication.

Therefore, all the arithmetic involving new angle calculation will be done within the GPU [Fig 5.1]. This leaves the step of moving the vertices using the calculated rotational angles. This, however, can not be done using the GPU by our approach. This is due to the fact that when moving a vertex belonging to an upper level branch it needs to know how the vertices in the branches below it moved as explained in 3.1.3. Therefore, moving the vertices using the calculated angles will be done within the CPU.



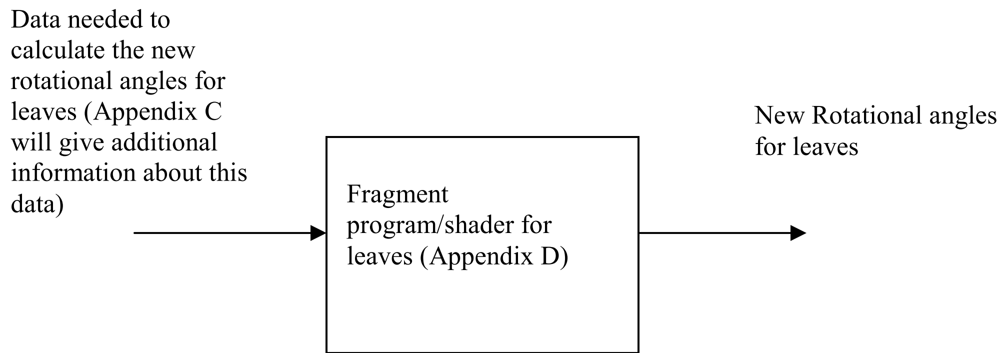


Fig 5.1 Using the GPU to calculate the new rotational angles for branches and leaves (It is possible to provide a single fragment program to handle both branches and leaves)

6. RESULTS

The main objective of our project was to port part of the calculations handled by the CPU into the GPU, resulting in an increase in the computational performance of the system. Such an increment in the computational performance would enable animating in real-time a number of 3D trees according to the wind patterns they are subjected to.

In order to prove the effectiveness of our approach, we first implement the tree animating program without transferring calculations to the GPU. This enables us to show how the performance of such an approach is inadequate as the number of trees increases. Next, we show how utilizing the GPU will increase the performance of the system. Utilizing the GPU to the extent that is possible with our approach can be divided into several stages. In each such stage we incrementally use additional features of the GPU, resulting in performance gains.

- Stage 1: Use the high performance memory of the GPU to store data
- Stage 2: Transfer to the GPU that portion of calculations associated with the new angles by which the branches and leaves need to be moved due to current wind forces (refer section 5.2), while utilizing the GPU high performance memory.

Frames Per Second (fps) was chosen as a suitable measurement of performance. Fps can be calculated by first noting the system time (`t0`) then allowing the program to render a certain number of frames (`numFrames`) and then noting the system time again (`t1`) [19].

fps = `(t1 - t0) / numFrames`

The reason for fps being selected as a suitable unit of measurement is because the value given by fps also gives an indication of how visually pleasing the image being rendered will appear. This can be explained in the following manner. The human eye retains an image for about one-tenth of a second after seen it [20]. Because of this effect, if the eye is exposed to a set of images of an object wherein each image object is drawn slightly to the right side of the previous image, this will be interpreted by the eye as the object

moving to the right direction. The number of images per second the eye should be exposed to in order for it to interpret the information from those images as a continuous motion differs. In our case 30 fps (or any number higher than) that will produce realistic looking tree motions while a lesser number of fps will result in jerky and unrealistic animation. Therefore, fps will serve both as a measurement of performance as well as a measurement of whether that performance is adequate to be interpreted as realistic or not.

Following are the important factors of our testing environment:

- We used a Pentium M processor 1.86GHz with an ATI Mobility Radeon x300 graphics card to obtain the following results.
- In order to maintain a consistent testing environment, it was decided to fix the number of branch levels in all trees at four (`iter`), and the number of subdivisions of a branch at three (`numD`).

6.1 Implementing the system without utilizing special features of the GPU

Number of Trees	fps
1	10
2	5
3	3
4	2

Table 6.1 Performance of the system
without utilizing special GPU features

At this point the trees will not be skinned and they will not contain any leaves. Each tree will be defined using 1952 vertices. However, it can be observed that the number of fps

(even with one tree) is below the 30 fps value that we deemed necessary for a smooth tree animation.

6.2 Stage 1: Use the high performance memory of the GPU to store data

Here the vertex information will be stored within VBOs as explained in section 2.3.4 enabling the use of the high performance memory on the GPU. At this point also the trees were not skinned and leaves were not added. Similar to the situation in section 6.1 each tree was defined using 1952 vertices.

Number of Trees	Fps
1	12
2	5
3	3
4	2

Table 6.2 Performance when utilizing the GPU high performance memory

It is interesting to note that the fps measurement values are quite consistent between Tables 6.1 and 6.2. However, this can be attributed to the fact that the amount of vertex data being stored within the GPU memory is quite small, and the bulk of the performance is in calculating the new branch and leaf angles. Therefore, it is not possible to observe a significant performance gain.

An option at this point was to increase the amount of vertices each trees is being defined with in-order to investigate the possible performance gain attributed to moving data into the GPU memory. However, given that the computational burden on the system was high within the present situation, we decided to forgo this option and further explore the features of the GPU with the intention of obtaining a significant performance gain.

6.3 Stage 2: Transferring calculations to the GPU while utilizing its memory

At this point, in addition to the GPU memory usage, we transferred a significant portion of the repetitive calculation (section 5.2) done up to this point within the CPU into the GPU. Similar to the above mentioned situations (6.1 and 6.2) the trees were not skinned, leaves were not added and each tree was defined using 1952 vertices.

Number of Trees	fps
1	105
2	60
3	40
4	31

Table 6.3 Using the high performance memory of the GPU and vertex shader and fragment shader to program the GPU

It should be noted that the performance figures in Table 6.3 are significantly higher than those in Tables 6.1 and 6.2. Therefore, the ability to program the GPU did indeed reduce the computational burden on the CPU and increase the performance as a whole.

In Fig 6.1, we have summarized the results of Tables 6.1, 6.2 and 6.3 using a chart. This helps to clearly show the significant performance gain experienced, when we used the GPU's memory and the ability to program it to carry out the necessary calculations.

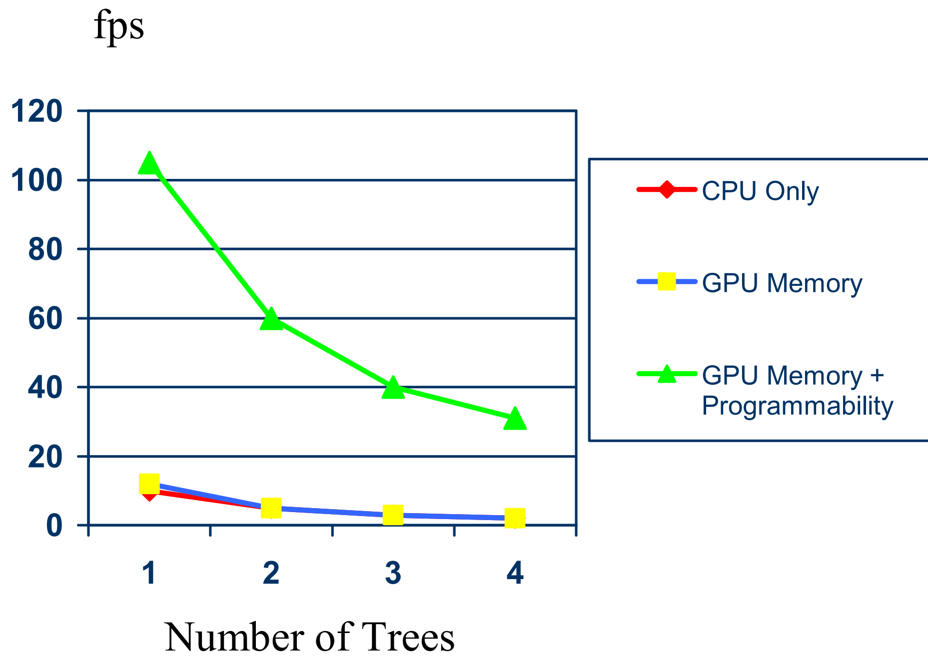


Fig 6.1 Summarizing the results in Tables 6.1, 6.2 and 6.3

6.4 Adding skinning to trees

Due to the significant performance gain experienced when a portion of the calculations were ported to the GPU, it is now possible to increase the computational complexity of the system, with the intention of increasing the realistic appearance of trees, while maintaining computational efficiency. Therefore, skinning (section 2.3) was introduced to the system.

Number of Trees	fps
1	90
2	45
3	36
4	31

Table 6.4 Performance when skinning is added to the trees

With the addition of skinning each tree structure is defined using 15616 vertices which form 29280 triangles. Such a level of computational complexity would definitely not have been possible if the CPU was handling all the calculations. However, as the performance figures reflect, the system can animate 4 trees above the 30fps limit that was selected as being necessary for realistic looking tree movement.

6.5 Adding leaves to the trees

As a final enhancement in appearance, leaves were added to the trees as explained in section 2.4.

Number of Trees	Fps
1	32
2	14
3	9
4	7

Table 6.5 Performance when leaves are added

This addition of leaves considerably increased the computational complexity of the program. At this point each tree is defined by 38656 vertices which form 40800 triangles. However, with the additional computational complexity, only one tree was rendered by the system at a satisfying rate for realistic tree animation. This situation can be altered by employing a computer with better resources.

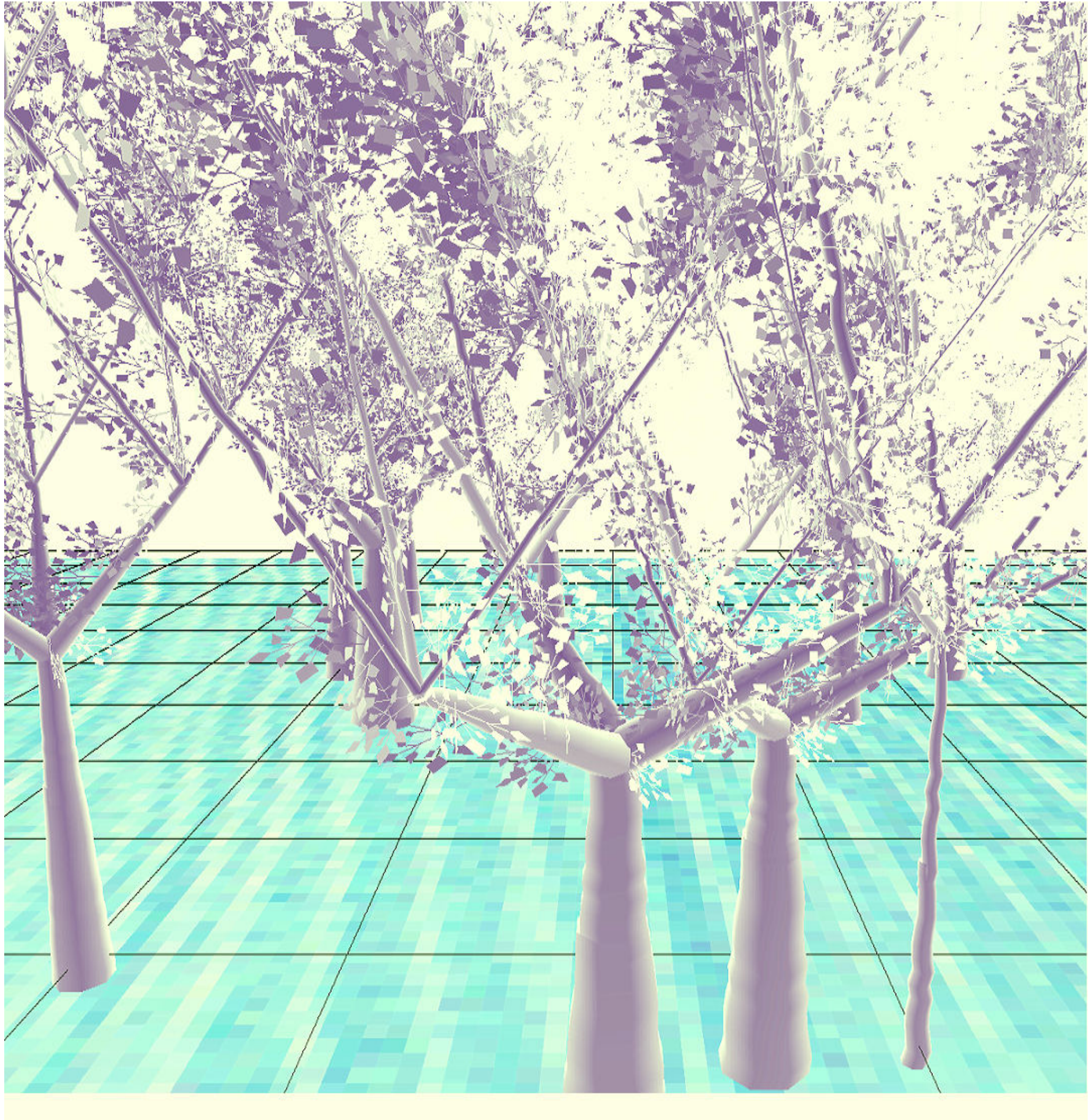


Fig 6.2 Multiple trees forming a forest

7. FUTURE WORK

Through the results we generated it is quite evident porting the calculations into the GPU is a successful approach for maintaining the computational efficiency while increasing computational complexity. This increment in the performance can be attributed to the fact that we were able to use the GPU's ability to handle computations in parallel, while at the same time make use of its enhanced algorithms to add and multiply floating point numbers.

In terms of future work that can be carried out we can focus on further utilizing the GPU's computational efficiency by transferring calculations which are presently handled by the CPU. Currently, when applying a transformation to a branch vertex it is necessary to know about the transformations vertices of connected branches in the lower levels were subjected to (section 3.1.3). Due to this dependency it was not possible to apply the transformations and move the vertices into their final positions using the GPU. However, if the transformation of a vertex can be expressed as a final value without dependencies then the calculations needed for moving vertices can be moved to the GPU so that it can be executed in parallel. This would significantly improve the computational efficiency of the system further allowing us to concentrate on improving the appearance of the animated trees. Apart from this, when using this system to create a forest environment the following features can be added to increase the computational efficiency and increase performance:

- Trees that are at a certain distance can be rendered using planer billboards [1] (instead of rendering its full 3D view) without harming the general appearance of the scene. This amounts to not calculating the new rotational angles of the branches and leaves of those trees, which results in saving considerably the computer's processing resources.
- For trees that are closer than above but which are directly not in the line of vision we can calculate only the new leaf rotational angles because doing the

entire set of calculations for both branches and leaves in such situations amounts to an inefficient use of resources.

REFERENCES

[1] L.McMillan and G.Bishop, "Plenoptic Modeling: An Image Based Rendering System", SIGGRAPH95 Computer Graphics Proceeding, pp.39-46, 1995

[2] Sakaguchi, Tatsumi and Ohya, Jun, "Modeling and Animation of Botanical Trees for Interactive Virtual Environments," *Symposium on Virtual Reality Software and Technology (VRST99)*, pp. 139-146, December 1999.

[3] S.-C. Chen, K. Zhang, and M. Chen, "A Real-Time 3D Animation Environment for Storm Surge," In Proceedings of the IEEE International Conference on Multimedia & Expo (ICME 2003), Baltimore, MD, USA, July 6-9 2003, pp. 705-708.

[4] Ota, S.; Tamura, M.; Fujita, K.; Fujimoto, T.; Muraoka, K.; Chiba, N. "1/f/sup /spl beta// noise-based real-time animation of trees swaying in wind fields" Computer Graphics International, 2003. Proceedings, Vol., Iss., 9-11 July 2003 Pages: 52- 59

[5] J.Beaudoin and J.Keyser, "Simulation Levels of Details for Plant Motion", EG/ACM SIGGRAPH Symposium on Computer Animation 2004 (SCA'04), Sept. 2004 Pages: 1-8

[6] F.K. Musgrave et al., Texturing and Modeling: A Procedural Approach, D.S. Ebert, ed., Academic Press, London, 1994.

[7] Ilya Shlyakhter , Max Rozenoer , Julie Dorsey , Seth Teller, Reconstructing 3D Tree Models from Instrumented Photographs, IEEE Computer Graphics and Applications, v.21 n.3, p.53-61, May 2001

[8] P. Prusinkiewicz, A. Lindenmayer and J. Hanan, “Developmental Models of Herbaceous Plants for Computer Imagery Purposes,” Computer Graphics, vol. 22, no. 4, Aug. 1988, pp. 114-150.

[9] P. Prusinkiewicz and A. Lindenmayer, The Algorithmic Beauty of Plants, Springer-Verlag, New York, 1990.

[10] Dalton, Colin. "SKELETON STRUCTURES." 25 May 2007 <http://www.cs.bris.ac.uk/Teaching/Resources/COMSM2003/kinematics.pdf>.

[11] Thomas Di Giacomo, Stéphane Capo and François Faure, “An Interactive Forest,” *Eurographics Workshop on Computer Animation and Simulation*, pp. 65-74, September 2001.

[12] Kanayama, Chisyun and Masuyama, Shigeru, “Animation of Botanical Tree Swaying,” *The Transactions of the Institute of Electronics, Information and Communication Engineers*, Vol. J80-D-II, No. 7, pp. 1843-1851, 1997, in Japanese.

[13] Kulkarni, Sandip D.; Minor, Mark A.; Deaver, Mark W.; Pardyjak, Eric R., “Output Feedback Control of Wind Display in a Virtual Environment” Robotics and Automation, 2007 IEEE International Conference on, Vol., Iss., April 2007 Pages:832-839

[14] Crow, T. S., 2004, Evolution of the Graphical Processing Unit, Master’s Thesis, University of Nevada, Reno.

[15] Perlin, Ken. "Noise and Turbulence." Ken's Academy Award. 30 May 2007 <http://mrl.nyu.edu/~perlin/doc/oscar.html#noise>.

[16] Mike Bailey and Steve Cunningham, "A Hands-on Environment for Teaching GPU Programming", SIGCSE 2007 Conference Proceedings, Covington, Kentucky, March 7-10, 2007, pp. 254-258.

[17] "A Facet of the nfiniteFX Engine ." Vertex Shaders . Nvidia. 30 May 2007 <http://www.nvidia.com/object/feature_vertexshader.html>.

[18] Montenegro, Rafael.; Montero, Gustavo.; Escobar, José María.; Rodríguez, Eduardo.; González-Yuste, José María., "An Efficient Package for 3-D Wind Simulation" University Institute of Intelligent Systems and Numerical Applications in Engineering, 2002, Pages:1 - 38

[19] "What do I need to know about performance?." OpenGL FAQ. Khronos Group. 9 Jun 2007 <http://www.opengl.org/resources/faq/technical/performance.htm>.

[20] "Science Gallery." General Information. The Manitoba Museum. 9 Jun 2007 <http://www.manitobamuseum.ca/sg_info.html>.

[21] Fatahalian K.; Sugerman J.; Hanrahan P., "Understanding the Effciency of GPU Algorithms for Matrix-Matrix Multiplication" Eurographics Association, 2004 Pages: 1-5

[22] Ohshima Satoshi; Kise Kenji; Katagiri Takahiro; Yuba Toshitsugu, "Parallel Processing of Matrix Multiplication in a CPU and GPU Heterogeneous Environment" Graduate School of Information Systems The University of Electro-Communications, Pages: 1-14

[23] Inman Matthew J. ; Elsherbeni Atef Z.; Smith Charles E., "FDTD Calculations using Graphical Processing Units", Center for Applied Electromagnetic Systems Research (CAESR) University of Mississippi, 2005 Pages: 1-4

[24] "Central processing unit." Definitions and Much More. 13 Jun 2007 <http://www.answers.com/topic/central-processing-unit>.

[25] Shirley, Peter, Michael Ashikhmin, Michael Gleicher, Stephen R. Marschner, and Erik Reinhard. Fundamentals of Computer Graphics. 2nd ed. Wellesley, Massachusetts: A K Peters, 2005.

APPENDIX A

ALGORITHM TO CALCULATE BRANCH ROTATIONAL ANGLES

Following is the Pseudocode for the algorithm that calculates new rotational angles (`angleXY` and `angleYZ` (3.1.1)) for branches one at a time.

1. Obtain the current rotational angles and store it in a temporary variable.
2. Obtain the 3D wind vector information at the start vertex of the branch.
3. Use the wind vector information to calculate the corresponding forces acting along the X, Y, and Z axis of the branch.
4. Convert the forces that are acting on the branch into deflections the branch will be subjected to within the 3D domain. Use the properties of the branch such as branch length, width, and thickness in the calculation so that the deflection will be different for each branch.
5. Use the rotational angles stored in step 1 with the deflection calculated in step 4 to obtain a new rotational angles
    a. Make sure that the new rotational angle is bounded within the minimum and maximum possible rotational angles.
    b. Make sure the new angles are small enough to ensure smooth movement.

APPENDIX B

FRAGMENT SHADER FOR BRANCH ROTATIONAL ANGLE CALCULATIONS

This is the fragment shader that will handle the calculations needed to update the branch rotational angles. Appendix A holds the pseudocode for the functionality carried out here. The comments inserted will explain the role played by the attributes and functions.

CODE STARTS HERE

```
uniform float branchL;  //Length of this particular branch
uniform float maxAngle; //Maximum angle by which this branch can swing
                        //in any direction
uniform float NUMB;     // This variable will help to adjust the speed
                        //of motion

//I decided to pass the noise values needed for calculations from the
//CPU as the GLSL functions for generating noise values were not
//working.
uniform float noiseVal1;
uniform float noiseVal2;
uniform float noiseVal3;

// These variables allow the access of the different textures
uniform sampler2D windInfoTexture;  //Wind information is stored here
uniform sampler2D angleInfoTexture; //The branch rotational angles are
                                    //stored here

uniform float elasticMod; //Elastic modulus of the tree
uniform float width;      //The width of the branch
uniform float thick;      //The thickness of the branch
uniform float spanL;      //The length of the branch

float angleXY = 0.01;  //Rotation angle in the XY-plane due to the wind
float angleYZ = 0.01;  //Rotation angle in the YZ-plane due to the wind
```

```
const float maxBranchLoad = 0.01;

//Function definitions
void branchMovement(float ang1, float ang2, vec3 wind);
void directBranchLoads(float newXLoad, float newYLoad, float newZLoad);
vec3 angleSetBranch();
float bSpringConst(float elasticM, float w, float t, float l);
vec3 updateBranch(float a, float b, float c);
float setBranchPartialAngle(float ang, float anyAngle);

//Temporary variables
vec3 angle;
float pass1, pass2, pass3;
float factor = 1.0;

float xBranchLoad, yBranchLoad, zBranchLoad;
float branchSpring;
vec3 endBranch = vec3(0.1, 0.1, 0.1);
vec4 pos;
vec4 posNew;


void main( void )
{
      vec2 texCoord = gl_TexCoord[0].st;
      vec3 texCoord3 = gl_TexCoord[0].stp;

    pos = texture2D(angleInfoTexture, texCoord);

    //Read the current angles from the texture
    vec4 posNew = texture2D(angleInfoTexture, texCoord);

    vec4 windL = texture2D(windInfoTexture, texCoord);

    angleXY = pos.x;
    angleYZ = pos.y;
```

```
    branchMovement(angleXY, angleYZ, vec3(windL.x, windL.y, windL.z));

    //Lets write these angles to a texture
    posNew = vec4(angleXY, angleYZ, windL.x, windL.z);


    gl_FragColor = posNew;
}

//This is the function which will tie the other functions' activities
//together
void branchMovement(float ang1, float ang2, vec3 wind)
{
      directBranchLoads(wind.x, wind.y, wind.z);
      angle = angleSetBranch();

      angleXY = angle.x + ang1;
      angleYZ = angle.z + ang2;

      //Make sure the angles do not exceed the limits
      if(angleXY > 0.0)
      {
            if(angleXY > maxAngle)
            {
                  angleXY = maxAngle;
                  factor = -1.0 * factor;
            }
      }
      else if(angleXY < 0.0)
      {
            if(angleXY < -maxAngle)
            {
                  angleXY = -maxAngle;
                  factor = -1.0 * factor;
            }
      }
```

```
	if(angleYZ > 0.0)
	{
		if(angleYZ > maxAngle)
		{
			angleYZ = maxAngle;
			factor = -1.0 * factor;
		}
	}
	else if(angleYZ < 0.0)
	{
		if(angleYZ < -maxAngle)
		{
			angleYZ = -maxAngle;
			factor = -1.0 * factor;
		}
	}
}


//Will set the direction loads.
//This is the function that will allow us to simulate the wind movement
//by calculating the directional forces acting on the branch due to
//the wind.
void directBranchLoads(float newXLoad, float newYLoad, float newZLoad)
{
	float n1, n2, n3;

	n1 = noiseVal1;
	n2 = noiseVal2;
	n3 = noiseVal3;

	if(n1 < 0.00001 && n1 > -0.00001)
	{
		n1 = 1.0;
		xBranchLoad = newXLoad + maxBranchLoad * n1;
	}
	else
```

```
{
	if(newXLoad == 0.0)
	{
		xBranchLoad = 0.0;
	}
	else
	{
		xBranchLoad = newXLoad + maxBranchLoad * n1;
	}
}

if(n2 < 0.00001 && n2 > -0.00001)
{
	n2 = 1.0;
	yBranchLoad = newYLoad + maxBranchLoad * n2;
}
else
{
	if(newYLoad == 0.0)
	{
		yBranchLoad = 0.0;
	}
	else
	{
		yBranchLoad = newYLoad + maxBranchLoad * n2;
	}
}

if(n3 < 0.00001 && n3 > -0.00001)
{
	n3 = 1.0;
	zBranchLoad = newZLoad + maxBranchLoad * n3;
}
else
{
	if(newZLoad < 0.00001 && newZLoad > -0.00001)
	{
```

```
                zBranchLoad = 0.0;
            }
            else
            {
                zBranchLoad = newZLoad + maxBranchLoad * n3;
            }
      }

}//end of directLoads()


//Will generate the final displacement angles due to a certain
//directional load
//We will visit this function after visiting directLoads() function
vec3 angleSetBranch()
{
      float defX, defY, defZ;
      float motionAngleX = 0.1, motionAngleY = 0.1, motionAngleZ = 0.1;

      branchSpring = bSpringConst(elasticMod, width, thick, spanL);

      defX = xBranchLoad / branchSpring;
      defY = yBranchLoad / branchSpring;
      defZ = zBranchLoad / branchSpring;

      updateBranch(defX, defY, defZ); //Lets update the new end
      position of the branch

      pass1 = (defX / branchL) * 0.7;
      pass2 = (defY / branchL) * 0.6;
      pass3 = (defZ / branchL) * 0.8;

      //Handles the value being passed to
      //arcsin() being greater than 1 situation.  =>
      //However if pass1, pass2, and pass3 values are always greater
      //than 1 then always the value 0.0 will get applied which is not
      //effective.
```

```
if(pass1 > 1.0)
{
      motionAngleX = asin(0.6);
      motionAngleY = asin(0.5);
      motionAngleZ = asin(0.3);
}
else if( pass2 > 1.0)
{
      motionAngleX = asin(0.6);
      motionAngleY = asin(0.5);
      motionAngleZ = asin(0.3);
}
else if( pass3 > 1.0 )
{
      motionAngleX = asin(0.6);
      motionAngleY = asin(0.5);
      motionAngleZ = asin(0.3);
}
else
{
      motionAngleX = asin(pass1);
      motionAngleY = asin(pass2);
      motionAngleZ = asin(pass3);

      //This bit of code will ensure each branch do not bend in
      //Uncharacteristic ways.  Angles are checked and adjusted
      //accordingly at this point. If we were to change the
      //angles someplace else the checks for those adjustments
      //should be done at that point.

      if(abs(motionAngleX) > maxAngle)
      {
            if(motionAngleX > 0.0)
            {
                  motionAngleX = maxAngle;
            }
```

```
				else
				{
					motionAngleX = -maxAngle;
				}
			}

			if(abs(motionAngleY) > maxAngle)
			{
				if(motionAngleY > 0.0)
				{
					motionAngleY = maxAngle;
				}
				else
				{
					motionAngleY = -maxAngle;
				}
			}

			if(abs(motionAngleZ) > maxAngle)
			{
				if(motionAngleZ > 0.0)
				{
					motionAngleZ = maxAngle;
				}
				else
				{
					motionAngleZ = -maxAngle;
				}
			}
		}

		return vec3(motionAngleX, motionAngleY, motionAngleZ);

}//end of function: angleSet()


//Updates the new position of a branch edge after wind movement
```

```
vec3 updateBranch(float a, float b, float c)
{
	endBranch.x = a;
	endBranch.y = b;
	endBranch.z = c;

	return vec3(endBranch.x, endBranch.y, endBranch.z);
}



//Will calculate the spring constant of the timber
float bSpringConst(float elasticM, float w, float t, float l)
{
	float bSpring;

	bSpring = (elasticM * w * t * t * t) / (4.0 * l * l * l);

	return bSpring;
}



//This method will setup the intermediate angles of the branch
//movement.  This will generate a smooth flow in branch movement.
//The reason I am passing two angles is so that I will have a starting
//angle from which I will move into another angle, instead of moving
//always from the original position given in the constructor.
float setBranchPartialAngle(float ang, float anyAng)
{
	float tempAngle = 0.0;
	float dif = 0.0;

	tempAngle = ang;

	if(tempAngle != anyAng)
	{
		if(tempAngle > anyAng)
		{
```

```
        dif = (tempAngle - anyAng) / NUMB;

        anyAng = tempAngle - dif;

        if(anyAng > 0.0)
        {
            if(anyAng > maxAngle)
            {
                anyAng = maxAngle;
            }
        }
        else if(anyAng < 0.0)
        {
            if(anyAng < -maxAngle)
            {
                anyAng = -maxAngle;
            }
        }
    }

    if(tempAngle < anyAng)
    {
        dif = (anyAng - tempAngle) / NUMB;

        anyAng = tempAngle + dif;
        if(anyAng > 0.0)
        {
            if(anyAng > maxAngle)
            {
                anyAng = maxAngle;
            }
        }
        else if(anyAng < 0.0)
        {
            if(anyAng < -maxAngle)
            {
                anyAng = -maxAngle;
```

```
                    }
                }
            }
        }
        return anyAng;
    }
```

APPENDIX C

ALGORITHM TO CALCULATE ROTATIONAL ANGLES FOR LEAVES

Following is the pseudocode for the algorithm that calculates new rotational angles (`motionAngleX, motionAngleY` and `totalRotaAngle`) for leaves one at a time. All the attributes used here are defined in (3.2.2)

6. Obtain the current rotational angles (`motionAngleX, motionAngleY` and `totalRotaAngle`) and store it in temporary variables.

7. Use Perlin noise values and maximum rotational angles to calculate three values for `motionAngleX, motionAngleY` and `totalRotaAngle`. (noise values will be in the range of [0, 1])

   (Consider calculating `motionAngleX`)
   `motionAngleX` = noise value * `maxMotionAngleX` * value in the range of (0, 1]
   Multiplying by a value in the range of (0,1] will help in making sure `motionAngleX` gives us values in the range of (0, `maxMotionAngleX`]

   (Formulas to calculate `motionAngleY` and `totalRotaAngle` are given in 3.2.2)

   The values calculated in this step for `motionAngleX`, `motionAngleY` and `totalRotaAngle` are only suggestions not the final values.

8. Depending on the values suggested for `motionAngleX, motionAngleY` and `totalRotaAngle` and the values stored in step 1 calculate the final rotational angles for each of them.

(Consider calculating `motionAngleX`)

a. If stored value for `motionAngleX` in step 1 is `angX1` > 0
b. If value suggested for `motionAngleX` in step 2 is `angX2` > 0
c. New `motionAngleX` = `angX1` + (`angX2` – `angX1`)
d. Make sure the new angle is within the range of
   (0, `maxMotionAngleX`]

APPENDIX D

FRAGMENT SHADER FOR LEAF ROTATIONAL ANGLE CALCULATIONS

This is the fragment shader that will handle the calculations needed to update the leaf rotational angles. Appendix C holds the pseudocode for the functionality carried out here. The comments inserted will explain the role played by the attributes and functions.

CODE START HERE

```
uniform float maxMotionAngleX; //Maximum angle by which this branch can
                               //swing
uniform float maxMotionAngleZ; //Maximum angle by which this branch can
                               //swing
uniform float maxRotationAngle;//This deals with the rotation the leaf
                               //does around the stem

//I decided to pass the noise values needed for calculations from the
//CPU as the GLSL functions for generating noise values were not
//working.
uniform float noiseVal1;
uniform float noiseVal2;

uniform float NUM;     // This variable will help in adjusting the
                      //speed of leaf motion
uniform float fact;   //Indicate the branch level

//These variables allow the access of the different textures
uniform sampler2D leafAngleTexture; //The branch rotational angles are
                                    //stored here

float motionAngleX = 0.01; //Rotation angle in the XY-plane due to the
                           //wind
float motionAngleZ = 0.01; //Rotation angle in the YZ-plane due to the
                           //wind
```

```
float totalRotaAngle = 0.01;//Rotation angle the leaf does around the
                           //stem

//Temporary Variable
vec4 pos;
float linkedFactor = 0.7;
vec4 posNew;
float helixAngle = 0.01;
float rotationAngle = 0.01;
float n1 = 0.01, n2 = 0.01;

//Function Definitions
void setMotionAngle();
void setRotationalAngle();
float setPartialAngle(float ang, float anyAngle );


void main( void )
{
    vec3 texCoord = gl_TexCoord[0].stp;

    pos = vec4(0.1, 0.1, 0.1, 0.1);

    //Read the current angles from the texture
    pos = texture2D(leafAngleTexture, texCoord);

    setMotionAngle();
    setRotationalAngle();

    motionAngleX = setPartialAngle( pos.x, motionAngleX );
    motionAngleZ = setPartialAngle( pos.y, motionAngleZ );
    totalRotaAngle = setPartialAngle( pos.z, totalRotaAngle );

    //Lets write these angles to a texture
    posNew = vec4(motionAngleX, motionAngleZ, totalRotaAngle,
                  noiseVal1);
```

```
    gl_FragColor = posNew;
}


//This is the function that will allow us to simulate the wind movement
void setMotionAngle()
{
      n1 = noiseVal1;
      n2 = noiseVal2;

      //Will handle to situation of noise values n1 = n2 = 0.0
      //For example if n1 = 0, then motionAngleX will become 0.0,
      //resulting in sudden change in value
      if((n1 == 0.0) && (n2 != 0.0))
      {
            n1 = 1.0;
            motionAngleX = maxMotionAngleX * n1 * fact;
      }

      else if((n1 != 0.0) && (n2 == 0.0))
      {
            n2 = 1.0;
            motionAngleZ = maxMotionAngleZ * n2 * fact;
      }

      else if((n1 == 0.0) && (n2 == 0.0))
      {
            n1 = 1.0;
            n2 = 1.0;
            motionAngleX = maxMotionAngleX * n1 * fact;
            motionAngleZ = maxMotionAngleZ * n2 * fact;
      }

      else
      {
            motionAngleX = maxMotionAngleX * n1 * fact;
```

```
            motionAngleZ = maxMotionAngleZ * n2 * fact;
      }
}//end of setMotionAngle()


//Will generate the rotational angle of the leaf
//This calculation is done according to the formulas given in [4]
void setRotationalAngle()
{
      float n3;

      n3 = (n1 + n2) / 2.0;

      rotationAngle = maxRotationAngle * n3;
      helixAngle =  linkedFactor * motionAngleX;

      totalRotaAngle = rotationAngle + helixAngle;
}


//This method will setup the intermediate angle of the leaf once a
//rotation angle is calculated.
//This will generate a smooth flow in leaf movement.
float setPartialAngle( float ang, float anyAngle )
{
      float tempAngle;
      float dif = 0.0;

      tempAngle = ang;

      if(tempAngle != anyAngle)
      {
            if(tempAngle > anyAngle)
            {
                  dif = (tempAngle - anyAngle) / NUM;

                  anyAngle = tempAngle - dif;
```

```
		if(anyAngle > 0.0)
		{
			if(anyAngle > maxMotionAngleZ)
			{
				anyAngle = maxMotionAngleZ;
			}
		}
		else if(anyAngle < 0.0)
		{
			if(anyAngle < -maxMotionAngleZ)
			{
				anyAngle = -maxMotionAngleZ;
			}
		}
	}

	if(tempAngle < anyAngle)
	{
		dif = (anyAngle - tempAngle) / NUM;

		anyAngle = tempAngle + dif;
		if(anyAngle > 0.0)
		{
			if(anyAngle > maxMotionAngleX)
			{
				anyAngle = maxMotionAngleX;
			}
		}
		else if(anyAngle < 0.0)
		{
			if(anyAngle < -maxMotionAngleX)
			{
				anyAngle = -maxMotionAngleX;
			}
		}
	}
```

```
        }
        return anyAngle;
}
```