Imperial College London
Department of Computing

Tree of Life Visualisation
By

Kai Zhong(kz12)

September 2013

Submitted in part fulfilment of the requirements for the degree of
Master of Science in Computing of Imperial College London

Abstract

The storage, searching and visualisation of big data has become an increasingly important issue in computer science. One technique for visualisation of huge hierarchies is the interactive fractal inspired graph(IFIG), which employs fractal geometry to place limitless amounts of information on a single page and show,or hide information by the simple actions of zooming and panning. This method has been employed(reference) for tree of life visualisation where it solved a significant outstanding problem in evolutionary biology: how can large evolutionary trees be visualised effectively for science, education and public outreach.

While the software runs on javascript and html5, the object of this project is to develop an android application based on his code. Compared with a personal computer, the calculation ability of a mobile device is relatively low and the RAM capability is more restricted. The project involves how to use multiple thread and temporary bitmaps to solve the performance issue, which greatly shorten the loading time as well as the response time of user interaction. When dealing with larger trees, because of the limitation of RAM, dynamical loading and freeing objects is employed in this project, such that the usage of RAM could be maintained under the limitation of an android application. Both the technique of multiple threading, using dynamic loading and freeing objects could be employed in the website software.

Acknowledgement

Contents

List of Tables

List of Figures

1 Introduction

1.1 Background

Visualising phylogenies is one of the fundamental tasks of evolutionary analysis. It has long been a great challenge to display huge trees. When the amount of species on the tree increases to millions, no display is huge enough to display the whole tree while maintaining all megadata(e.g., name, population stability, conservation status) of each species. Even if a screen large enough is accessible, it is unpleasant for users to look for information on such a large screen.

One way to solve this issue is the interactive fractal-inspired graph(IFIG), which includes all species as well as all their megadata. The key concept of the IFIG method is to place all the data on one page such that users could navigate between different scales by simply zooming in and out (http://www.onezoom.org). Therefore, users can't see all information of all species at one time. Instead, when the scale is big enough, users could see the whole picture of the tree of life but with few details of each species. When users zoom the tree, more details of the selected area will be displayed while other species might go out of the screen.

The interface of the OneZoom is analogous to Google Earth. When using google earth, one can find a location by zoom from a start page of the whole globe, recognizing familiar landmarks at different scales along the way(e.g., continents, countries, regions, and towns)(TODO: add reference). Equivalently, OneZoom enables the user to zoom smoothly to the species they want to find. For example, one can find human beings by zooming in along the clades of animals, vertebrates, mammals and primates.(reference) The hierarchy dataset is represented by using fractals. Fractals are typically self similar patterns, where self similar means they are "the same from near

as from far". This means the shapes of branches and nodes remains similar at different scales of zoom.(TODO: add reference) Hence, it could ensure the amount of visual elements on a screen remains relatively constant. Moreover, fractals simplify the representation of huge hierarchy dataset. All levels of data are represented in the same way. So inserting or deleting data can be automatically adopted by the software in the tree representation.

TODO: Duncan suggested to add more background about previous weak and perhaps something more on fractals and self similarity. Besides, Duncan also suggested to add some background about android development environment.

1.2 OneZoom Android Application

The IFIG method was only implemented using javascript and html5. The aim of this project is to develop an android application of it. The application is based on the javascript code written by Dr. James Rosindell.

When the application was developed, the first issue that arose is that the loading time of a tree is over long. This problem also occurs in the website software, though it is not very obvious due to faster processor of personal computer than mobile device. The fluency of user interaction is another issue. There were apparent delays when users interacted with the application. Both of these issues has been solved and details of that will be discussed in Chapter 4.

As the data gets larger, more memory needs to be allocated. As the memory one android application could be allocated is quite small, the application sometimes crashed when drawing the tree of tetrapods which has over 22 thousand species. This is also a potential issue for the further development of the software on OneZoom website when the size of tree grows to over million species. A more detail description of this problem the solution of it will be discussed in Chapter 5.

1.3 Structure of this Report

Chapter 2 gives introduction to the software that will be built on. It includes what interactions users could have with this software, the basic structure of the software, what methods are used to increase the performance of the software as well as some functions that will be inherited in the mobile application. Chapter 3 mainly talks about some changes in the implementation of the node class and how the application listens to user interactions.

Chapter 4 discusses ways to improve the performance of the application. It includes shortening loading time as well as making user interaction more fluent. When the dataset gets larger, the application will hit memory limitation. Chapter 5 gives a description of this issue and also the solution for it.

Chapter 6 goes through other functionalities that the application has implemented.

2 OneZoom Website Software Introduction

This chapter gives a brief introduction to the OneZoom website software, on which the whole project has been built. It talks about how the raw data is organized and the member variables of the node object built based on the raw data. Then it explains the criteria for selecting elements to be drawn. These two sections is highly related to the memory issue that will be discussed later. Then an introduction to the software working flow would be given. It is highly related to the performance issue in Chapter 5. Lastly, it would talk about how to use the software from a user perspective.

2.1 Data Processing

2.1.1 Raw data format

The raw data is a string which is in Newick format. In Newick format there are nodes, branch lengths and brackets. For instance, if two species A and B have a common ancestor node C 1.5 million years ago the Newick string looks like this:

(A:1.5, B:1.5)C.

Then if species D and species C have a common ancestor E 1 million years ago, the Newick string becomes:

((A:1.5,B:1.5)C:1,D:1)E.

If these group of species have a further ancestor, then this string would be took as a whole and replace A in the first expression to form a larger Newick

string. By recursive doing this, a string of all the biological data could be built.

Building a tree from the string is just the opposite as building the tree. For instance, given the second string, the node E out of the brackets would be first taken out. Then the remainder of the string would be split to two children.

$$\text{"((A:1.5,B:1.5)C:2,D:3.5)E"} = \text{"("} + \text{"(A:1.5,B:1.5)C:2"} + \text{","} + \text{"D:3.5"} + \text{")"} + \text{"E"}$$

Therefore, a node object E with two children would be built. Its first children need to be further initialised while its second children would be node D. For a larger dataset, building a tree follows the same steps. First creates the node out of the outer bracket, then split the data inside the bracket into two parts and set each part being one of the children of the outer node. Then continue initialising its children by the same steps.

Finally, for any Newick string, a binary tree starting from a root would be built.

2.1.2 Node object

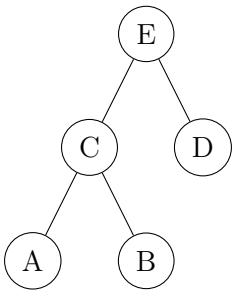
Basically, the nodes in a Newick string could be categorized into two types, link node(node with children) and leaf node(node without children).

For each leaf node in the Newick string, it contains a species name and the megadata e.g. a node named: Tachyglossus_ aculeatus{Short-beaked Echidna_ LC_ S} means that it's the animal 'Tachyglossus aculeatus' (latin name) with common name Short-beaked Echidna and redlist status LC(which means least consern) and population stability S(which means stable). While for the link nodes, it does not contain information of its redlist status and its population stability in the raw data. These two information in the link nodes are statistic of the redlist status and population stability of all of their children.

Besides redlist and population stability, the richness of link nodes are also statistic of their children, while the richness of a leaf node is always 1.

Another megadata that cannot be accessed directly from the raw data is

the age of link nodes. The age of the root need to be inferred from its children. Take the previous string as an example, the node hierarchy can be represented as below. The age of node C is 1.5 million years from either child A or B. While the age of node E is 3.5 million years from child C or D.



The data talked above are information about a species, e.g. the name of a species, the richness of a species. Except these information, node objects also contain information about drawing an element. More specifically, these information can be divided to two parts. The first part tells the software where to draw the elements like bezier start, control and end points and how big should the elements be drawn like ithe branch line width, circle radius. The second part tells the software the bounding box of a node. Each node has two kind of bounding box, a bigger one representing a bounding box of itself and all of its children and a smaller one representing a bounding box of the node itself.

2.2 Ways to decrease the amount of elements being drawn

For a tree big enough, not all of the nodes would be drawn at any given time. There are two criteria for selecting elements to be drawn. The first is to compare the size of a node with a threshold. When the size of a node is smaller than the threshold, then the node should not be drawn, so does all of its children. There are also some other thresholds. Both leaf nodes and link nodes could show text. The text could appear or disappear, displaying in greater details or in less details. All these are controlled by the relative threshold and the size of these nodes. The second criteria is to test whether the bounding box of a node is within the screen. Here we use the bounding

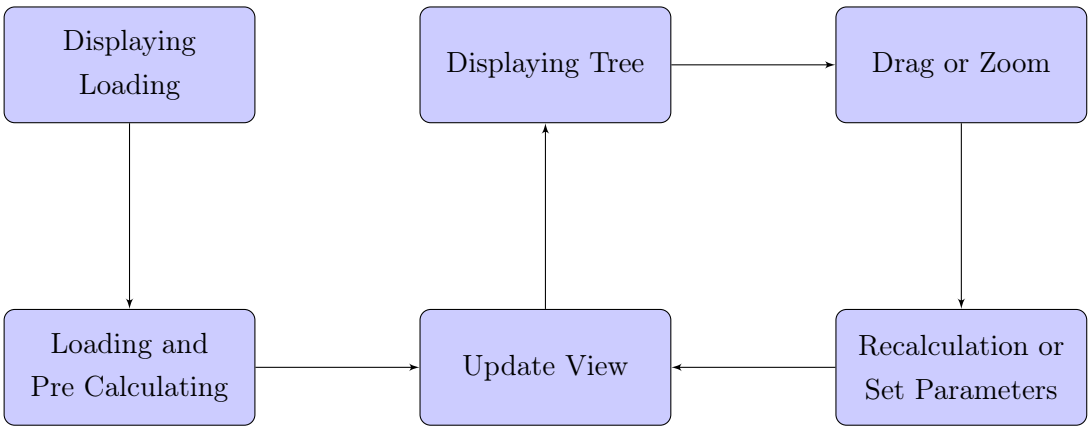
box containing only the node itself. If the node is in the bounding box, then it should be drawn, otherwise it should not be drawn.

2.3 Software Working Flow

The software first draws a loading page since it might take some time for the full tree being loaded and the loading page tells users that the software is still responding. The loading time increases linearly with the size of the species, which could become unacceptable for larger trees. A reasonable solution for shortening loading time can be found in chapter 4: Performance Issues.

After drawing the loading page, the software starts creating objects representing tree nodes from data. This is followed by initializing some node information, like the age of some nodes, which cannot be got directly from data but has to be calculated from their children nodes. Then pre calculation is called to calculate the Bezier curve parameter of each node such that the relative position of each node is known.

When pre calculation is over, the software will call a method to update the view and display the whole tree. From then on, user actions like tap and drag are listened and methods for recalculating the position and scale of node objects will be called. When recalculation is done, the tree will be updated.



2.4 How to use the software

Basically, the OneZoom software allows users to explore the tree of life in a completely new way: it's like a map, everything is on one page, all you have to do is zoom in and out(reference). When a user zooms in, some nodes become visible and more details will be shown on nodes. On the other hand, when a user zooms out, some nodes become invisible and some details will disappear in order to provide the user a neat view.

For example, in the figure of the tree of mammal, the tail of the tree is covered by a signpost rodents. If we zoom in this area, we can see that more signposts appears telling us more details of the rodent family. If we continue zoom in the mouse-like rodents, we can see that nodes that was invisible in the first two figures emerging.

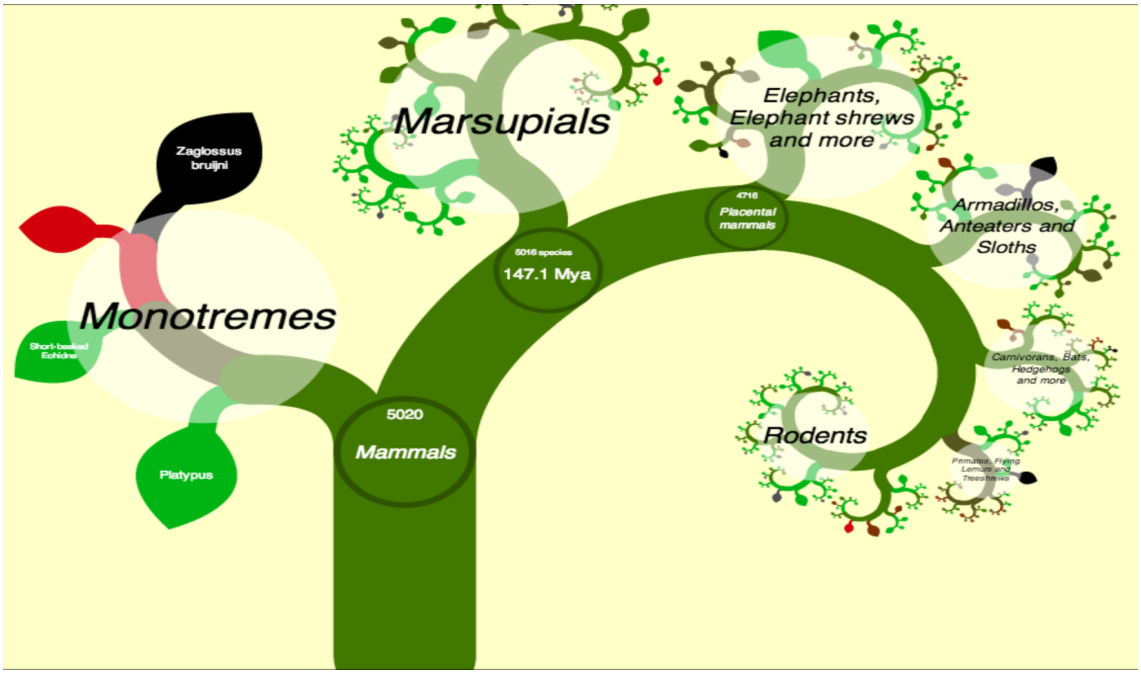


Figure 2.1: Tree of Mammal

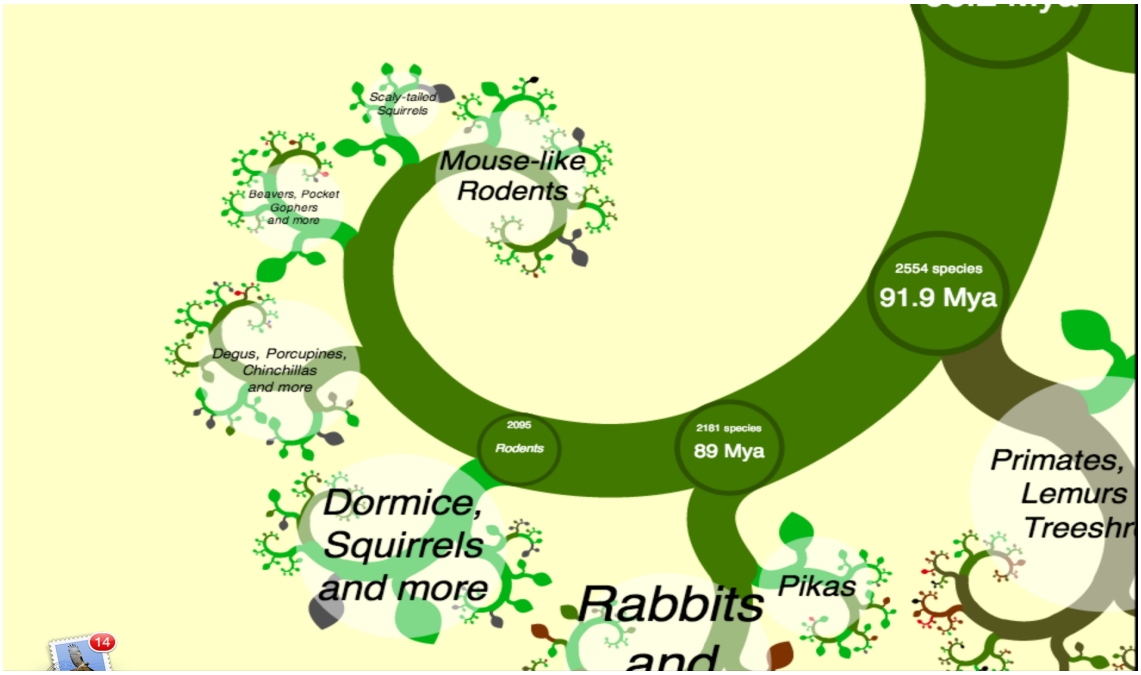


Figure 2.2: Tree of Rodent: Zoom In the Rodent Branch

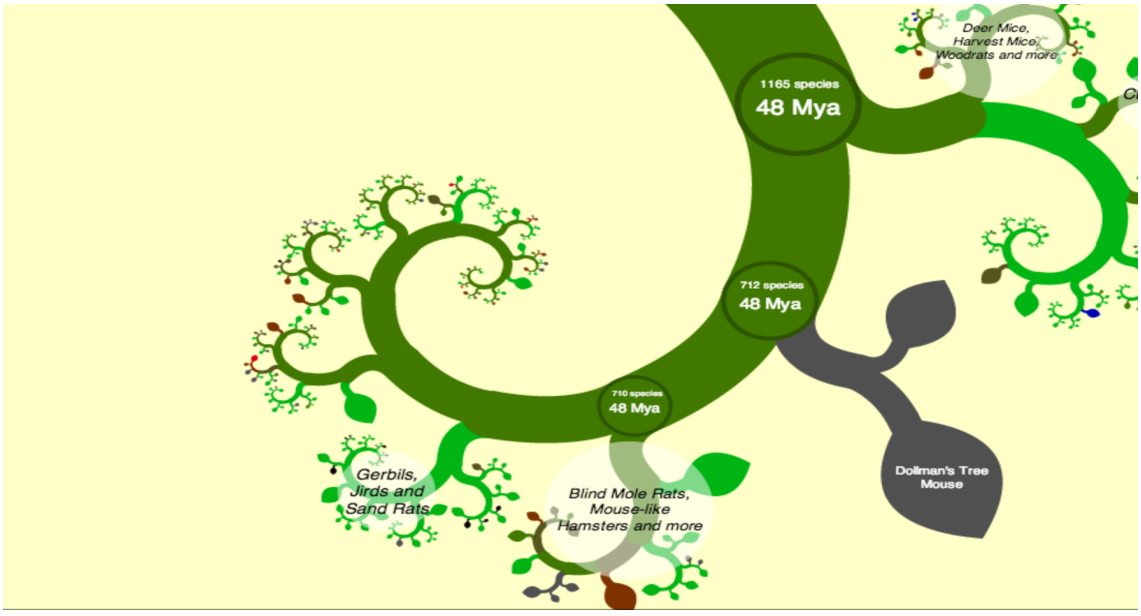


Figure 2.3: Zoom In the Mouse Like Rodents Branch

3 OneZoom Android Application Introduction

This chapter will describe changes of implementation from the website software to android application. More details concerning how to improve the performance of the software and how to break the memory limitation for larger trees will be discussed in Chapter 4 and Chapter 5.

3.1

The website software use one class representing both link node and leaf node. Instead of using only one class, this application uses the original node class as a super class that takes charge of how to calculate positions of a node, how to get information of a node, and LeafNode, CircleNode(link node) as subclasses of the original class that take charge of how to draw themselves. Basically, the tree contains two type of nodes, one is the node with children nodes, the other is the node without any children. The nodes with children will create CircleNode objects, while the other ones will create LeafNode objects.

3.2 User Operation Listener

The application now supports three kinds of user interaction: using one finger to drag the tree, double tap to zoom in, using two fingers two zoom in or zoom out. Drag and double tap are detected by inheriting GestureDetector.SimpleOnGestureListener class, while using two finger to zoom is detected by inheriting ScaleGestureDetector.SimpleOnScaleGestureListener class. Then in the subclasses of these listeners, onScroll, onDoubleTap and

onScale methods are overrode such that it could call the tree to be recalculated and the view to be updated.

```
        // event when using two fingers to scale
        @Override
        public boolean onScale(ScaleGestureDetector detector) {
                scaleFactor = detector.getScaleFactor();
                scaleFactor = Math.max(0.5f, Math.min(scaleFactor,
                    5.0f));
                drawview.zoomin(scaleFactor, detector.getFocusX(),
                    detector.getFocusY());
                return true;
        }

        // event when scroll occurs
        @Override
        public boolean onScroll(MotionEvent e1, MotionEvent e2,
            float distanceX, float distanceY){
                drawview.drag(distanceX, distanceY);
                return true;
        }

        // event when double tap occurs
        @Override
        public boolean onDoubleTap(MotionEvent e) {
                drawview.zoomin(e);
                return true;
        }
```

4 Memory Issue

4.1 Problem description

Memory is an important factor that needs to be considered under the circumstances of a quite large dataset and relatively resources restricted android development environment.

As the dataset grows greater, the amount of memory needed for creating nodes also increases. By using chrome developer tools, a snapshot of the heap memory usage could be get. For the mammals tree which has 5020 species, there are 6.0 megabytes allocated in heap for creating these nodes. 12.2 megabytes and 29.1 megabytes are allocated for birds tree which has 9993 species and for tetrapods tree which has 22822 species respectively. The maximum dataset which representing all species contains about 2 million species. To build such a tree, the chrome browser need to allocate approximate 2.4 gigabytes to 2.6 gigabytes, which is apparently not acceptable in most modern personal computers. The maximum amount of memory available for an android application is more limited than for a website software. More specifically, for each application running on Sumsang Galaxy S3, the maximum amount of memory available is 256 megabytes. This means the largest tree that could be built theoretically in the device has about 213,000 species, which is about one tenth the size of the full biology tree.

Tree Name	Richness	Total Heap Usage	Node Heap Usage
Mammals	5020	8.3Mb	6.0Mb
Birds	9993	14.4Mb	12.2Mb
Tetrapods	22822	34.1Mb	29.1Mb
Bacteria	408135	N/A	N/A

Table 4.1: Heap usage of tree with different amount of species in Chrome

Not only heap memory but also stack memory will increase as the number of species grows. When initialization begins, the website software creates the tree from the root. Then it recursively creates its children if it is not a leaf node. The parameters passed down to create its children are the dataset of the children, which is a string containing all megadata of its direct children and all further children. Therefore, for a leaf node which has a depth of n, its megadata would be copied in stack for n times during initialization. The greater amount of species a tree contains, the deeper its average depth will be. For the most balanced binary tree which contains N species, the average depth K of the tree should be:

$$log_2 N + 1 < K < log_2 N + 2$$

In the worst case, where the binary tree is totally unbalanced, the average depth K of the tree should be:

$$K = N/2$$

Take the largest tree released on the website tetrapods as an example. The data size of the tree is 1.5 megabytes. In the best case when the tree is well balanced, the average depth of the tree is about 16. Thus each megadata representing one species would be pushed into the stack for averaging 16 times. Therefore, the stack memory is expected to increase for at least 24 megabytes.

The largest dataset that currently could be obtained is the bacteria tree. The data size of the tree is about 11 megabytes, while the total species of the tree is 408135. Hence in the best case, the average depth of the tree would be 20. This multiply the size of the dataset is the memory that would be allocated during initialization: 220 megabytes. The bacteria tree currently could not be loaded using chrome because maximum call stack size exceeds.

Below are two tables gives comparison of total memory usage between different browsers and total memory usage right after initialization and a long time after initialization.

Tree Name	Richness	Chrome	Firefox
Mammals	5020	53.1Mb	10.39Mb
Birds	9993	58.5Mb	20.00Mb
Tetrapods	22822	80.7Mb	45.6Mb
Bacteria	408135	N/A	340.0Mb

Table 4.2: Memory usage of different trees between Chrome and Firefox

Tree Name	Richness	Right after Initialization	Long time after initialization
Mammals	5020	53.1Mb	93.3Mb
Birds	9993	58.5Mb	111Mb
Tetrapods	22822	80.7Mb	148Mb

Table 4.3: Memory usage of different trees between right after initialization and a long time after initialization in Chrome

4.2 Overview of the solution

The solution of the issue is to dynamically add and delete nodes. As mentioned in the introduction of the software, only when a node is inside the display area and when the size of the node is big enough to be seen, the node will be drawn. The approach this application takes is to select the elements whose larger bounding box(the bounding box containing the node itself and all of its children) are inside the screen and the size of the node is greater than the visible threshold. Therefore, only parts of the tree will be built. Then when the user interacts with the view, the application will allocate memory to the nodes that should be built and deallocate memory from the nodes that are not visible.

When the application starts or when the user interacts with view, the application begins to test what elements should be drawn. It starts from creating the root node. Then it test whether the root node should be drawn based on two criteria: whether its larger bounding box is inside the screen and whether its size is greater than the drawing threshold. If it satisfy the conditions, then continuing build both of its children and test their children

using the same criteria. The nodes that satisfy the criteria will recursively create its children and test them until their children does not satisfy the test condition or they don't have further children.

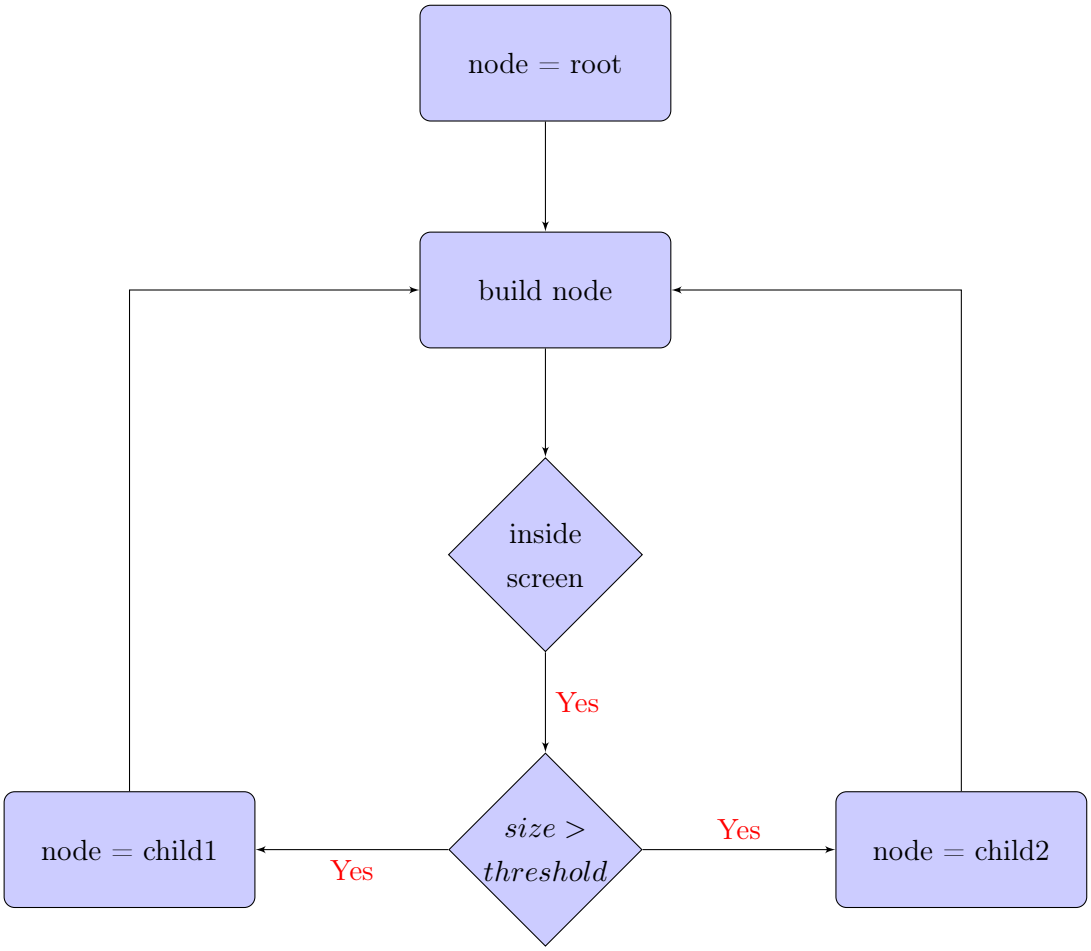


Figure 4.1: Work flow of building part of the tree

Suppose the screen area is the yellow box in the picture below. The blue box is the larger bounding box of the root. It represents the boundary of the root node and of all of its children. Apparently, the root node is inside the screen and its size is big enough to be displayed. Therefore continue to create both children node and test them.

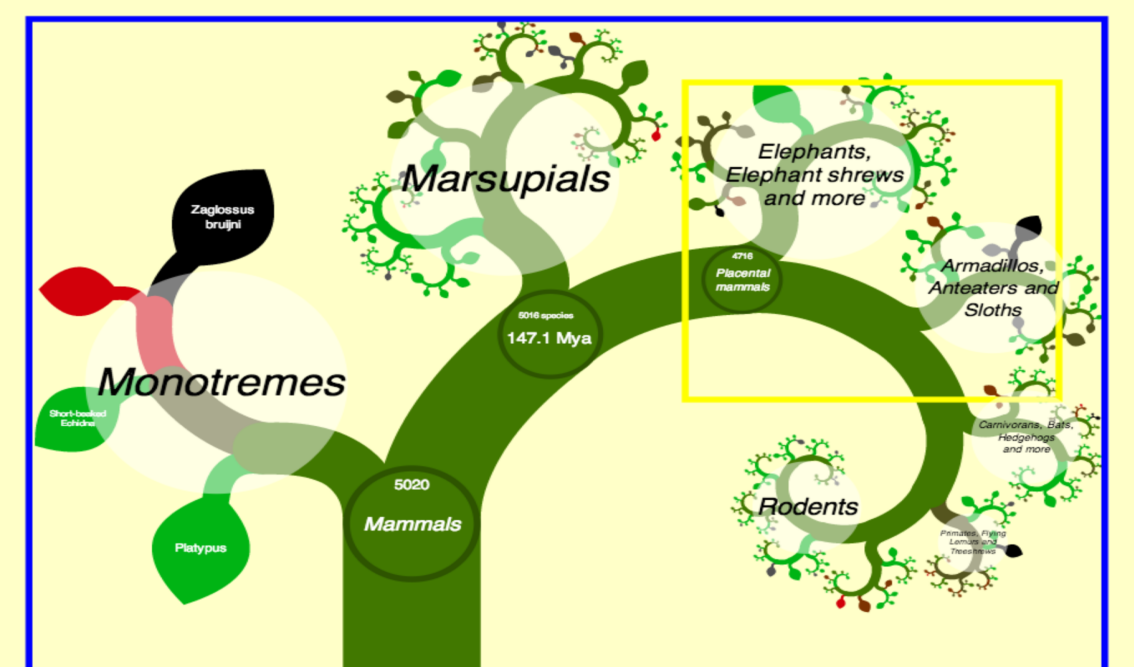


Figure 4.2: Using Bouding Box to select branch

In the second picture, the bounding box of child1(blue box) is out of the screen, while that of child2(red box) is still in the screen(Here it is assumed that all children on the left hand side are child1). Therefore, the application would continue to build the children of child2 and test them. While the children of child1 will not be further checked.

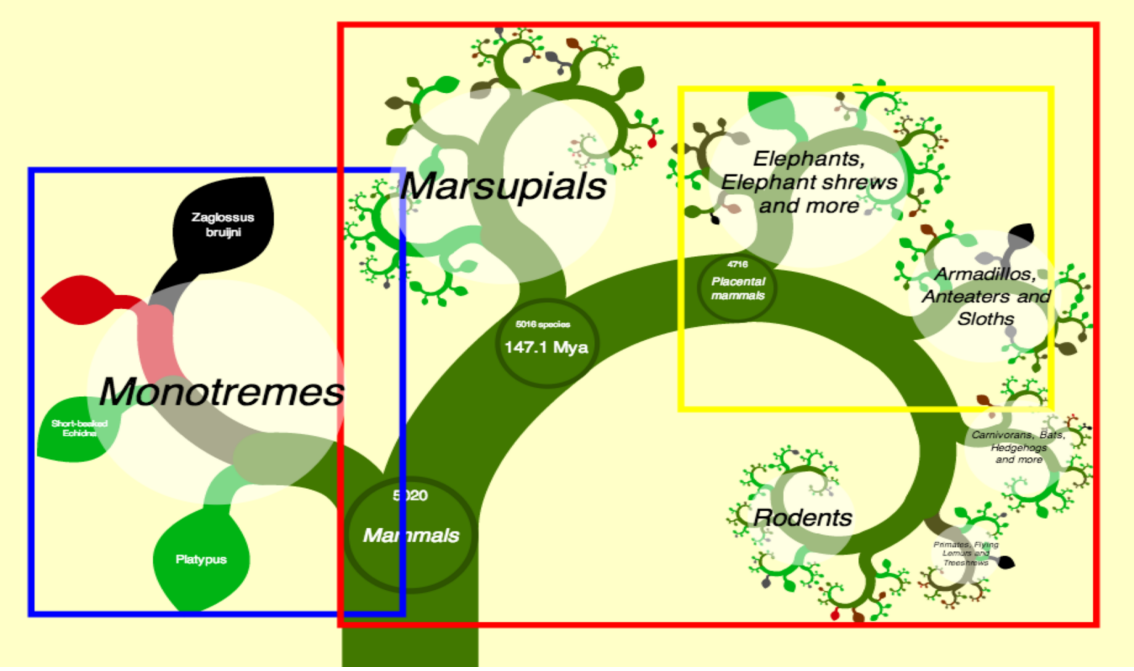


Figure 4.3: Bounding box of child1 and child2 of root

Further building will continue until it reaches the leaf node or the node is out of the screen or the node is too small to be displayed.

4.3 Create new data

Since only part of the tree would be built, the information of the nodes that relies on its children, like richness, age, could not be calculated using the raw data. Hence, new dataset need to be created containing those information. For example, the dataset discussed in Chapter 2:

((A:1.5,B:1.5)C:1,D:1)E

should be changed as:

((A[1_0],B[1_0])C[2_1.5]:1,D[1_0])E[3_2.5]

Inside the square bracket, the first number represents the richness of a node and the second number represents the age of a node.

Except for the richness and age, the calculation of outer bounding box also relies on its children. Therefore, the bounding box is also expected to be appended in the raw data. However, in order to avoid the raw data becomes too long and too complex, a database is created to record those information. It also contains other information like the ratio of a node, the x position and y position relative to the root of the tree. Those data is used to test whether the node should be drawn.

The database also includes the id of both children. When the node pass the test, then it could build and test its children according to their id. In order to build a node, we need to access data of that node. Instead of storing the whole data string into the database, the application records the start and end position of the node in the data string. What's more, as the data for the children could also be accessed by referring the start and end position in the database, the application don't need to pass down the whole data string as parameters. Instead, the application create new nodes by passing down the children id. Therefore, each substring representing a node in the raw data would be created only once, which greatly reduce the use of stack memory.

▼MAMMALS	table	
ID	field	INT PRIMARY KEY
RICHNESS	field	INT
HXMIN	field	FLOAT
HYMIN	field	FLOAT
HXMAX	field	FLOAT
HYMAX	field	FLOAT
XVAR	field	FLOAT
YVAR	field	FLOAT
RATIO	field	FLOAT
ANGLE	field	FLOAT
CHILD1	field	INT
CHILD2	field	INT
CUTSTART	field	INT
CUTEND	field	INT
COMPARERICHNESS	field	INT

Figure 4.4: Database columns

Again, because of memory size limitation, the application can't build a database containing all nodes. Instead, only for the nodes whose richness

or whose parent richness is greater or equal to a threshold, the node information would be recorded in the database. For the nodes whose richness is smaller than the threshold, the start and end position of its data will represent itself plus both of its children. For bacteria tree, the threshold is 500. The table of bacteria contains 19543 rows, which is about 2.5% the total amount of nodes in the original bacteria raw data.

In the following paragraph, the node in the database whose richness value is smaller than threshold will be called mini root node. When this node pass the test of displaying, the application will not continue build and test because it does not have enough information about its children. Instead, the application will continue building all of children directly from the string.

4.4 Dynamic adding and deleting nodes

This section gives more details about how the application adds and deletes nodes. During the process of adding and deleting nodes, two lists, running list and non initialization list are used.

When the application starts, it builds the root node and add it into non initialization list. The running list is initialized as empty list. Then each time user interacts with the view, the recalculation begins.

The recalculation begins in the root node. The first step is to test the node position, size and its richness. There are four types of possible test results. Test result 0 means the node should not be drawn. Test result 1 means the node should be drawn and its a leaf node. Test result 2 and test result 3 means the node should be drawn and they are link nodes. The difference is that test result 3 means the richness of the node is smaller than the threshold, that it's a mini root node.

If the test result is 0, the recalculation should stop here. Otherwise if the result is not 0 and if the node is not in the running list, the node will be added into the running list. Specifically, if the result is 1, the recalculation will also end, while if the result is 3, all children of the mini root node will be built and no further check is needed. If the test result is 2, then the children of this node will be built by referring records from the database and repeat

the steps mentioned above.

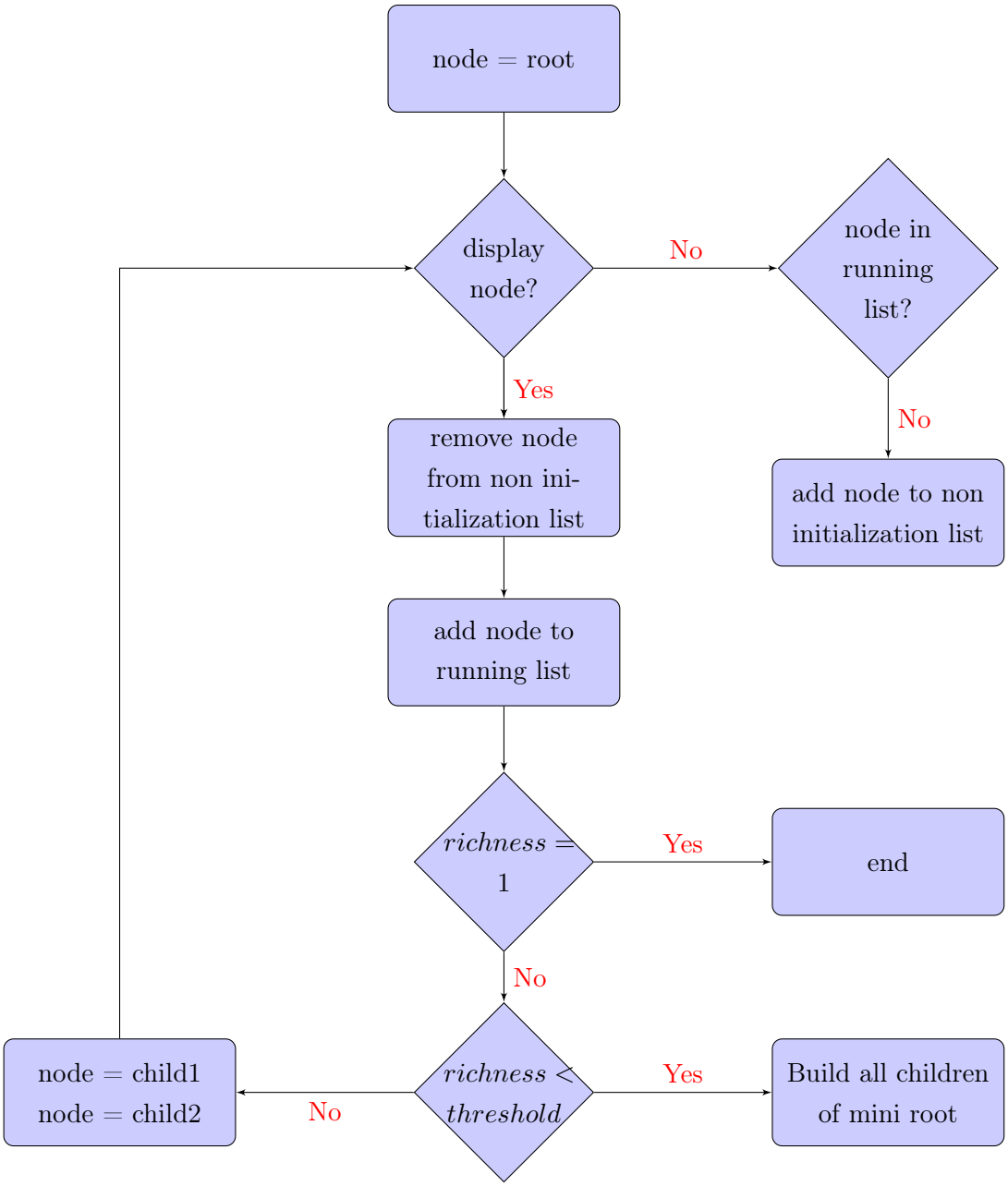


Figure 4.5: Work flow of adding nodes on the tree

From the picture we could see if the node is in the running list and it should not be displayed, the node and its children will remain in the running list.

The node is not removed from the list and get deallocated for performance reasons. Logically, the initialization list represents the nodes whose children are not built. If the node is moved from running list to non initialization list, then all of its children also need to be removed from running list and get deallocated. Besides, during the recalculation in the future, these nodes are possible need to be built again, which also cost great amount of time. Therefore, by remaining nodes in running list, the application could save time from deallocating nodes as well as allocating nodes.

As the user do more and more interaction, the remaining memory may be not enough for allocating new nodes. Therefore, after adding new nodes is done, the application will check whether there are enough space left(10 megabytes for current application). If the memory is not enough, then deleting will start.

The first step of deleting is to test the possibility of the nodes in the running list being displayed in the following user interactions. In general, the closer the node's bounding box is to the screen and the bigger the node's size is, the higher possibility the node will be assigned. Suppose that d is the distance between bounding box and screen, s is the size of nodes, T is the visible threshold, WH is the average of the width and height of the screen. Then the possibility of the node is:

$$Possibility = Integer.Maximum - P(d) - P(s)(d > 0, s < T)$$

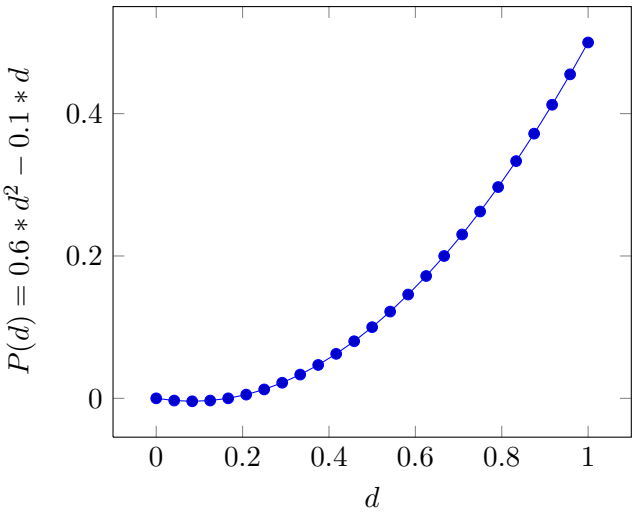


Figure 4.6: Possibility with distance

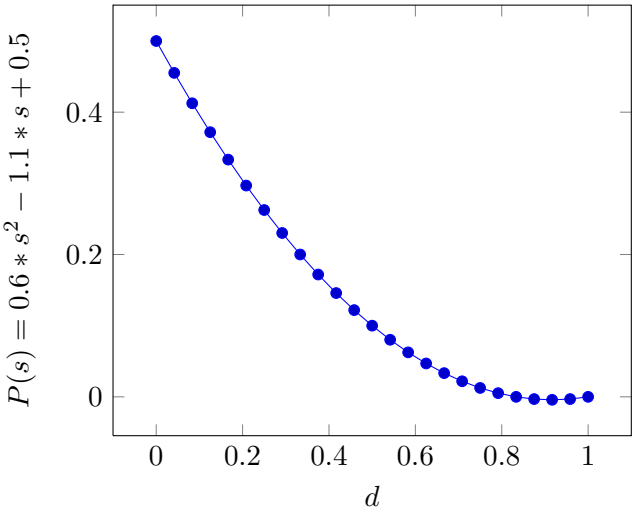


Figure 4.7: Possibility with size

5 Performance Issue

Two problems greatly affect the performance of this application. One is over long loading time, the other is slow user interaction response time. Both of the problems are largely influenced by the processing capacity of a mobile phone processor as well as the size of the dataset.

5.1 Loading Time

Before the phylogenies tree first gets drawn, the application needs to load data from a string and create objects for the logical units in it, which represents a species or a group of some species. Then pre calculation which involves all the nodes on the tree gets called in order to calculate the relative position of each node. Both the cost of loading data and the cost of pre calculation is linear. Hence, the loading time of a phylogenies tree is expected to increase linearly with the size of the data. By using an older version of the application, we can test the loading time of different phylogenies trees on a Sumsang Galaxy S3 device. When loading the mammal tree which has 5020 species, the loading time is 1761 millisecond, while loading the bird tree which has 9993 species and tetrapods tree which has 22822 species, the loading time increases to 3683 milliseconds and 8177 milliseconds respectively. The increase of loading time with the increase of species can also be observed in the website software. More details could be seen in the table below. As the ambition of this project is to build a tree of all existing and existed species, which has approximate 2 million species, the loading time of that tree could be intolerable.

Tree Name	Richness	Android	Chrome	Firefox	Safari
Mammals	5020	1761ms	319ms	239ms	501ms
Birds	9993	3683ms	660ms	538ms	1452ms
Tetrapods	22822	8177ms	1951ms	1087ms	2972ms
Bacteria	408135	N/A	N/A	17016ms	59684ms

Table 5.1: Loading Time on Android and website version before optimizing

In order to shorten the loading time, several bitmaps of those trees are stored as resources. When a user selects one of the tree, the corresponding bitmap would be drawn on canvas. In the meantime, another thread would be created to load data and do pre calculation. The main UI thread and the calculation thread share a flag by which the view object knows whether the pre calculation is done or not. When the pre calculation is on progress and the user drags or zooms on the tree, the view object will use the built in method of canvas class to scale or translate the tree. When pre calculation is finished, the calculation thread would send a message to the main UI thread so that it would invalidate itself.

Essentially, in the previous design, users spend time waiting for the data to be loaded and calculated. In the later design, the tree is drawn using bitmap even before the tree is created. Therefore, from a user perspective, the loading time of any tree could be reduced to unnoticeable.

5.2 User Interaction

Another factor that could affect user experience is the slow response time for user interaction. When user drags or zooms on the view, the view listener would receive a set of drag or zoom messages. For each message it receives, the application is supposed to recalculate which nodes should be display and create and add those nodes which has not been created yet into the drawing array. Then it will call the view to update itself. Since both the speed of recalculating and the speed of redraw nodes are more slower than the speed of generating drag or zoom messages, user could feel great delay waiting for these two procedures finishing.

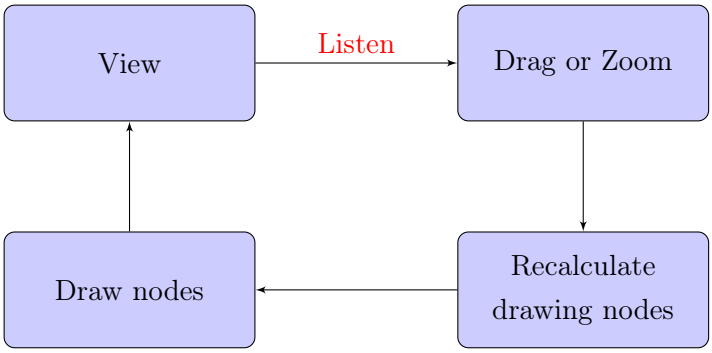


Figure 5.1: Previous working flow of main UI thread

5.2.1 Using thread and message queue to discard extra mouse move message

In order to reduce the response time, the extra finger move messages should be discarded. Another thread and message queue is used to solve this problem. When a user triggers a set of move actions, a set of recalculation messages will be sent from the main UI thread to the calculation thread. In the calculation thread, when it catches a recalculation message while there are other recalculation messages in the message queue, then the current recalculation message will be discarded. Otherwise, the recalculation method will be called. Hence, when there are several recalculation messages being sent, only the last one will be processed. When the recalculation is over, the calculation thread will send message to invalidate the view.

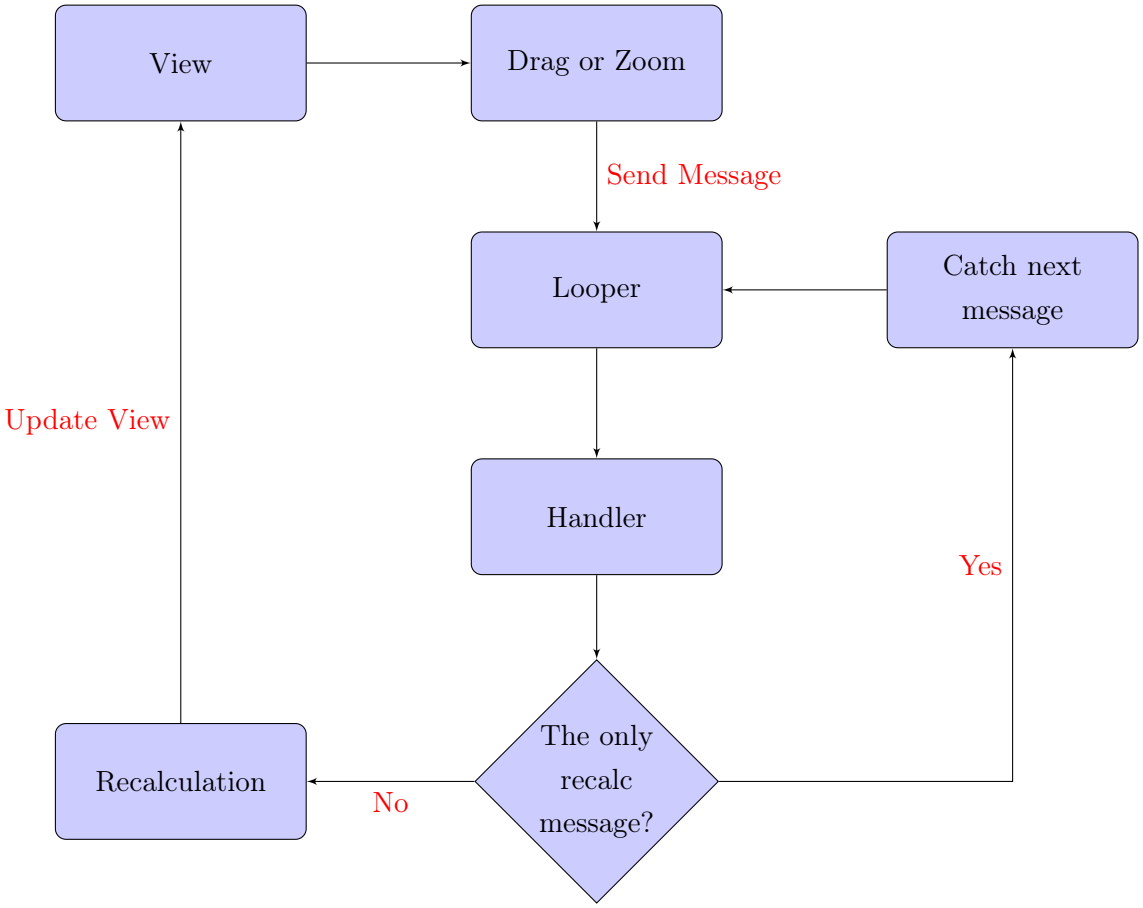


Figure 5.2: How recalculation thread deals with message

5.2.2 Using idle thread to prepare nodes

Currently, the calculation thread is active only when the user interacts with the view. The time users waiting for the view to draw nodes are decided by how long the thread would take to add new nodes. An improvement of the thread implementation was made by taking full use of its idle time.

There are two arrays(TODO)

5.2.3 Using cached bitmap to update view before calculation is done

Though the number of calculation messages being processed would be reduced to a minimum number, users would still feel a bit of delay in refreshing the view since the view only gets updated after the recalculation is done, which may take a bit of time. Moreover, as only the last recalculation message in the message queue gets processed, the tree will be placed back to the position where the last mouse move message was triggered. This could result in inconsistent display.

This problem is solved by using a cached bitmap to update the view. When the application observes a user operation, it will not only send a message to another thread, but also update the view. Then in the onDraw method of the view class, it would test whether the recalculation is done. If it is done, it will draw the tree objects, otherwise it will translate or scale the bitmap stored previously to give users a continuous display. The size of the bitmap is actually greater than the size of the screen. When the user drags or zooms out the view , the part that was out of the screen would be moved into the screen. If the bitmap is the same size as the screen, then user would see a compressed picture on the screen. When the recalculation is done, it will send message to update the view. From a user perspective, when he or she drag on the view, he or she would get a consistent display. However, if the user zooms in, he or she would get an unclear picture first, since scaling on bitmap is only scaling on pixels. However, a precise picture of the tree would replace the unclear one soon.

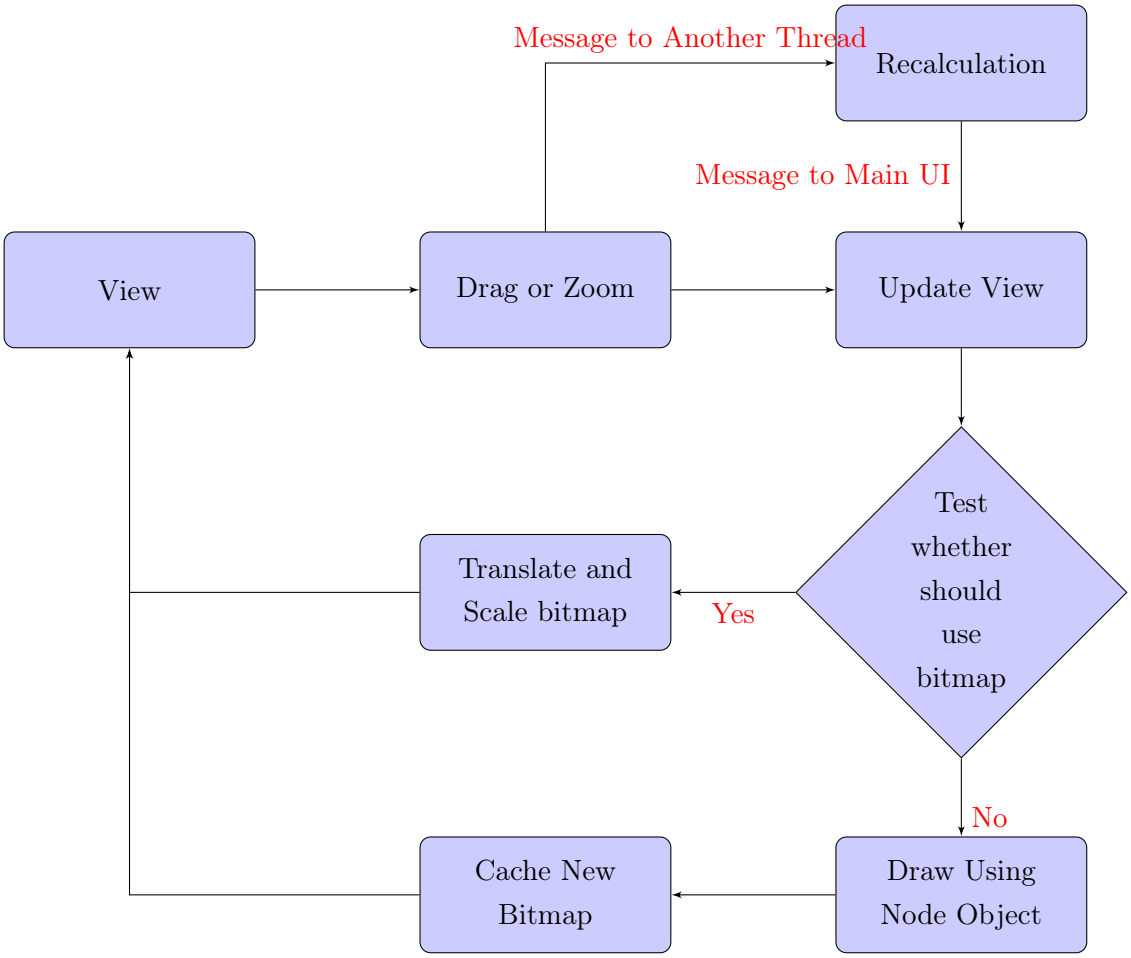


Figure 5.3: How to update view

5.2.4 Test use bitmap or do actual drawing

Both listening to user actions, drawing nodes, drawing bitmap are processed in main UI thread. Therefore, when the application is drawing bitmap or drawing nodes, the application is not listening to user interaction at the same time. The time spent on drawing bitmap is nearly 0 milliseconds, while the time spent on drawing nodes is fluctuate between 50 to 150 milliseconds. This means that there will be obvious delays if the view gets updated by drawing nodes while user is still interaction with the views.

Therefore, the application need to avoid drawing nodes during user interaction. This is done by set a flag named duringInteracion when the view

detects action down and reset the flag when the view detects action up, which suggests the user's finger is off the screen.

There is one more thing needing to be tested for whether to use bitmap . When the processor of a mobile device is switched to handle recalculation message, it could quickly discard all but the last recalculation message and process it. However, while the recalculation is being done, user could continue operating on the mobile device. For example, the calculation thread may be switched while the user is moving his or her finger on the screen. As a result, when the calculation is done, the position of the tree object is not where the user's finger points to. If the user's finger is off the screen now, the application would draw using nodes, then the user could see the tree gets dragged back. In order to solve this bug, we need to check whether the recalculation is triggered by the nearest user operation. Two counters are created for the check. One is incremented when a recalculation message is generated. The other is incremented when a recalculation message is received in another thread. When two counters are equal, then the last recalculation message is under processing or has been processed. To prevent the view gets redrawn using the node tree while recalculation is undertaking, a Boolean variable is set to true when recalculation message is sent and is set to false when recalculation is done. Together with two counters, it could ensure the tree only gets drawn by the tree object when the recalculation has been done and it is corresponding to the last user operation.

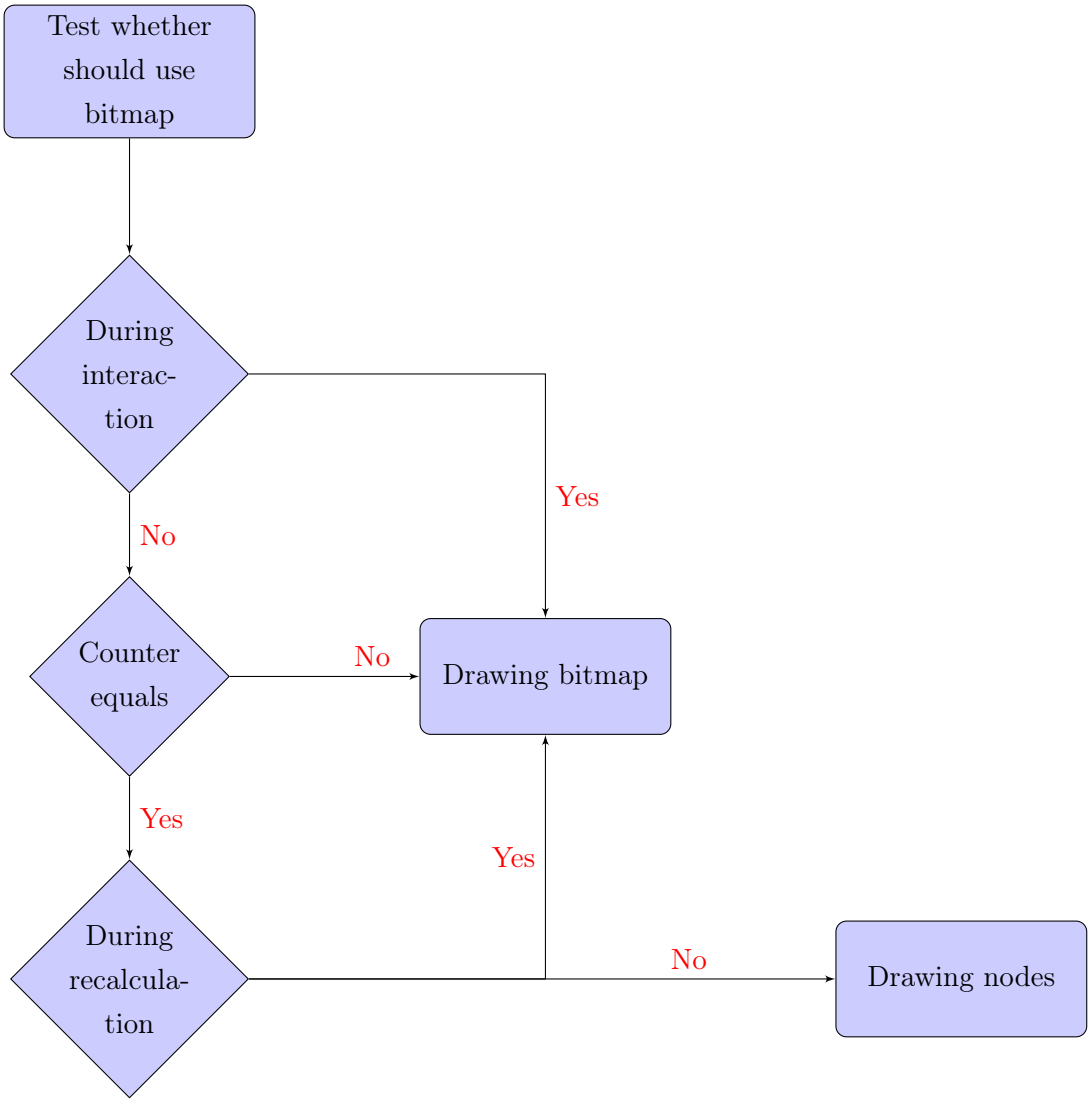


Figure 5.4: How to test whether use bitmap to redraw

5.2.5 Avoid duplicate drawing nodes

When the user moves his or her finger off the screen, the application would reset the duringInteraction flag so that the last time the view should draw nodes instead of draw elements. The application indeed draw nodes after user ends interaction. However, drawing nodes is triggered for several times instead of only once. Apparently, it is not necessary since by using counters and testing whether during recalculation the nodes in the drawing list should

be the corresponding to the last user interaction. Since the user's finger is off the screen, the nodes in the drawing list would be the same. Hence, it is not necessary to draw nodes for several times.

This issue is caused by the invalidate message generated from the calculation thread. When user interaction ends, the application would reset duringInteraction flag and update itself. If another thread ends its calculation and sends message to invalidate the view at this time, then the view will get updated for two times both under the circumstance of duringInteraction flag being reset. Hence, it would result in extra drawing nodes.

5.2.6 Draw less detail first

In order to update the view by drawing nodes as soon as the user moves their fingers off the screen, the application would increase the threshold of displaying nodes when interaction begins. Therefore, less nodes would be added and drawn in the drawing list. It would shorten the time for recalculating as well as drawing. 5 seconds after the view is updated with high threshold, the view will send another recalculation message to another thread to calculate again with more details. If during this 5 seconds, user interacts with the view, the result of this recalculation won't be drawn since the counter in view has been updated. If user has no interaction with the view, then after user could see the view being updated with more details. The reason for setting 5 seconds is that user is very likely to do another interaction followed by the previous interaction. If the interaction is taken during drawing nodes with more details, the view then can't listen user interaction. Therefore delay could occur.

From a user perspective, he or she could get the view rapidly updated after their interaction with the view. Then 5 seconds later, more details would be drawn on the screen.

6 Functionality

Bibliography

[1] It is more convinient and faster to use `bibtex` instead of writing your bibliography manually.

[2] You can even use a tool like `jabref` to manage and maintain your database of references.