Virtual reality visualisation of phylogenetic trees

Thomas Hemery[1], Jamie Twycross[1]

1 Department of Computer Science, University of Nottingham, Nottingham, Nottinghamshire, UK

Abstract

The demand for efficient, versatile methods of displaying phylogenetic information is ever-increasing. The aim of this project was to develop a novel system for displaying phylogenetic trees in a Virtual Reality (VR) environment, building on work that was previously done in the field and bringing it together under this evolving medium. The resulting software was developed within Unity for the Vive Focus mobile VR platform to provide the user with an intuitive and unique approach to viewing phylogenetic data.

This document will aim to explain the process of implementing the software and demonstrate how it functions as a "proof of concept" of how VR can be utilised to improve user understanding of the complex hierarchical information contained within phylogenetic trees. To achieve this aim we provide the user with multiple methods of viewing tree data, some of which introduce novel methods of displaying trees, with the aim of reducing the effort in learning key information from phylogenetic trees on evolutionary closeness and gene inheritance through time.

The goal of this project was to produce functional software to demonstrate the many applications of VR within the phylogeny field, in the realisation of this goal we have developed a novel algorithm for displaying complex trees in VR that is a true "first of its kind". The work done here shows how the application of VR to visualising phylogenetic trees and the field of data visualisation in general is an area worthy of a great deal more research and attention than it currently receives.

Introduction

Phylogenetic analysis is a cornerstone of modern biological research. The assessment and understanding of phylogenetic trees and the evolutionary relationships between species not only allows clear classification of organisms but is a vital part of learning how current genetic sequences came to be and how they might change in the future according to Emery [1]. Phylogenetic trees displayed in two dimensions as branching diagrams represent the defacto standard for displaying such evolutionary information, allowing the viewer, at least with smaller trees, to quickly glean important information on the evolutionary heritage of a species or gene of interest.

Representing clearly the information contained within large, complex phylogenetic trees meanwhile is an open ended and complicated issue. Constructing complex phylogenetic trees from the vast amounts of genetic data currently available is a significant challenge already and the effort is wasted if the resulting information cannot be meaningfully understood. The phylogenetics field as a whole is "experiencing an increasing but poorly met requirement for software supporting the advanced visualisation of phylogenetic trees" [2]; this is a problem created by the consistently increasing level of data available and the need to display, in a human comprehendible form, the wildly complex trees that can be constructed from it.

While a great deal of work has been put in to developing tools and methods to build phylogenetic trees there has been little done in the way of creating new or novel methods to display the data, beyond small variations of the standard two-dimensional tree representation. Further development and research in this area is sorely needed as the standard tree representation often fails to make evolutionary distances clear in larger trees; it becomes virtually impossible to easily trace branches from the root of the tree or sub tree to the target leaf nodes to determine genetic relationships as a "high cognitive load" is "required for tracing lineages to their common ancestral nodes to determine lineage and clade relationships" (Both according to Waese, Provart, Guttman [3]).

The inherent drawbacks of tree diagrams at larger scales motivate the production of novel, hierarchical data visualisation methods that reduce said cognitive load and enable the user, at a glance, to discern important information. A complete detachment from this model however could also have drawbacks as the tree method is widely adopted and implemented and is easily understandable to most audiences. For that reason, this visualization tool aims to enhance the tree diagram by simplifying information extraction while keeping the mental effort of translating understanding from trees to this new data visualisation method minimal. For example, we aim to allow the user to tell quickly which nodes are closely linked using a topography system, and then to cross reference between this and the underlying tree structure to more rapidly glean key information. Along with this we provide a system for displaying meta-information about each node, that the user can view at run time to glean extra understanding of the species and genes behind the phylogenetic structure.

Developing such a visualisation system within a VR environment also represents an area of research that has very little previous academic attention. A search for academic material brought up only the work of Forghani, Vasev and Averbukh [4] who researched using a MATLAB based system to produce a phylogenetic tree viewer in VR; their work will be discussed in more depth in the related works section of this document, but their implementation represents a more limited and traditional tree-like representation than was the aim of this research. It is our hope that the natural intuitiveness of utilising VR technology should translate to viewing trees using this system, and that the ability to physically explore trees as real-world objects should benefit the user's understanding / comprehension of the information contained within a phylogenetic tree of interest.

Implementation

Core Components

The project was built on top of the following set of core components: the Vive Focus headset and controller, the Vive wave SDK, the game engine and development environment Unity and C# scripts used at runtime to define the system's behaviour. The following subsection will aim to explain the reasoning behind each of these choices.

The Vive Focus VR System

The Focus (Vive Focus) is an ergonomic, portable alternative to the flagship HTC Vive VR system. Instead of relying on external light towers for position tracking and a powerful computer to drive the visual system the Focus instead has an on-headset method of position tracking and uses internal hardware to drive the display, running a modified Android operating system. As a result of its portability it is not capable of quite the same level of visual complexity as its alternative, but the high level of mobility and ease of use coupled with the lack of a need for complex graphics within this project make it a very suitable platform.

The Focus system also has an included controller that can be used to control VR applications with a greater degree of ease than, for example, a Google Daydream or Google Cardboard device could offer – neither of which have controllers as standard, giving the Focus a significant advantage over its smartphone-based competitors.

In terms of features the Focus system is as follows:

- Tetherless VR headset running a modified Android operating system with:
  - On board computation
  - 6 degrees of freedom position and rotation tracking
- Wireless controller with:
  - 3 degrees of freedom rotation tracking
  - Touchpad input
  - Trigger and button inputs

The Vive Wave SDK

The Vive Wave SDK is tool suite for developing mobile VR applications. The software supports various VR platforms including the Vive Focus system that this project will be based on. Most importantly, the SDK also contains plugins for development using either Unity or Unreal Engine, each popular and widely adopted game engines. The SDK is utilised to allow a Unity project to be installed and run on the device.

Unity

Unity is a full featured game engine and development tool. Although this project has no connection to gaming, the rich options for 3D graphics, cross platform support and visual processing given by modern game engines makes using one seem like a sensible decision; as mentioned previously, two game engines are currently supported by the Vive Wave SDK: Unity and Unreal Engine. For this project Unity was chosen as: it has good native support for android, it has excellent support for three-dimensional graphics, it provides excellent usability, and I have personal development experience with it. Craighead J. and co provide a similar set of reasons for utilising Unity to develop their robotics simulation environment SARGE and within this highlight that it “comes with complete documentation with examples for its entire API” [5], which is a highly beneficial inclusion for development efficiency.

While it would be possible to utilise just the Vive Wave SDK and an android development environment like Android Studio, the features made available by Unity (a complex component system for handing in app objects, inbuilt C# scripting support, a detailed 3D rendering system with preview capabilities, native support for android and – with the SDK – the Vive Focus headset) make it a much preferable choice.

Projects within Unity are composed of groups of “GameObjects” organised in a hierarchical tree structure, each with associated attached components; through this system Unity supports an entity component system that nicely separates classes and data and helps to enforce good programming practice. The Unity UI is explained in Fig 1.

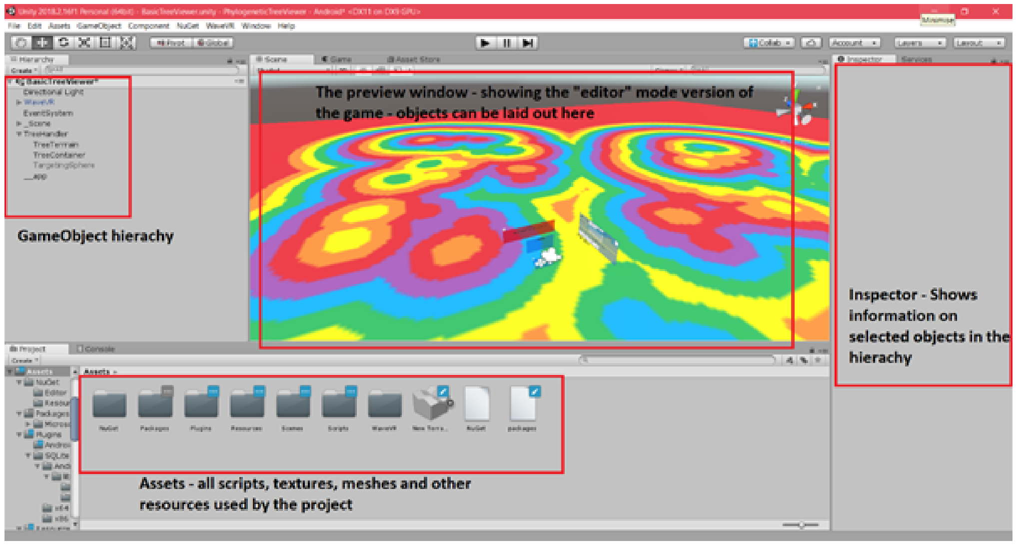

Fig 1. The Unity UI. Key areas are highlighted.

Algorithmic Components

To achieve a feature rich visualisation system several core algorithms were developed; these will be explained in turn in this subsection.

Newick Parser

The first aspect of the project to be considered was the development of a parser capable of reading in Newick files and producing a coded representation of the contained tree. Fundamentally this algorithm broke down into two stages, tokenization and parsing. The tokenizer sub-algorithm was responsible for taking the input text file as a string and converting it into a sequence of valid tokens or rejecting the input as being lexically incorrect. The parser would then take the sequence of tokens and convert it into a final tree, or reject it as being syntactically incorrect.

The tokenizer's implementation was fairly straightforward as all tokens are single character length – the algorithm simply compares the head of the input text with the available tokens and saves the correct token to the output list. In the event that a character that is not recognised as a token is read, the algorithm can simply assume that a word is being read – this will either be an edge weight or a node label, and the result is a word token that can be written to the output list in the same way as all other token. For these reasons the Tokenise algorithm will run on any input string without the need to generate any errors; all error handling can be performed in the parsing algorithm instead.

The algorithm's pseudocode implementation is as follows:

```
input   : A string target containing a newick descriptor
output  : A list of tokens tokens represented as strings

valid_tokens ← list of valid newick symbols
tokens ← empty list of strings
word ← ""
reading_word_flag ← false
foreach character c in target do
    if valid_tokens contains c then
        if reading_word_flag then
            append word to tokens
            reading_word_flag ← false
        end
        append c to tokens as a string
    else
        if not reading_word_flag then
            reading_word_flag ← true
            word ← ""
        end
        append c to word
    end
end
```

Algorithm 1: Tokenizer

The list of tokens created by the tokenizer can be directly handed to the parser for parsing and generation of a final tree representation. The parser loops through the generated tokens and, depending on the selected character, takes several actions culminating in the generation of a full tree structure in memory as one "Tree" object. Because of the deeply nested nature of trees and their Newick representations, the parser relies on the usage of a "node stack" that stores freshly created nodes; nodes are then popped from this stack and named in reverse order, depth first (from leaf nodes back to the root node).

The algorithm's pseudocode implementation is provided below as a helper function and a main algorithm:

```
input  : A Tree object tree, a name label name, a node stack node_stack, a depth
         value depth
output : A named node top_node from the node stack

Function popAndName(tree, name, node_stack, depth):
    pop top_node from node_stack
    add top_node to tree
    if name == "" then
        set top_node label to ""
    else
        set top_node label to name
    end
    set top_node depth to depth
    if node_stack is not empty then
        peek parent_node from node_stack
        if parent_node is not none then
            add top_node as child of parent_node
            set top_node parent to parent_node
        end
    else
        if top_node is not root of tree then
            SYNTAX_ERROR
        end
    end
    return top_node
```

Algorithm 2: PopAndName function

```
input  : A list of tokens tokens represented as strings
output : A Tree object created_tree

created_tree ← empty Tree
node_stack ← empty Stack
created_tree root ← empty Node
push created_tree root to node_stack
depth ← 0

expect_name_flag ← true
last_named_node ← none
is_label_flag ← false

foreach token in tokens do
    if token length == 1 then
        if token first character == '(' then
            depth ← depth + 1
            push new node to node_stack
            expect_name_flag ← true
        else if token first character == ')' then
            depth ← Depth - 1
            if expect_name_flag then
                last_named_node ← popAndName (created_tree, none, node_stack,
                  depth)
            end
            expect_name_flag ← false
        else if token first character == ',' then
            if expect_name_flag then
                last_named_node ← popAndName (created_tree, none, node_stack,
                  depth)
            end
            push new node to node_stack
            expect_name_flag ← true
        else if token first character == ';' then
            if expect_name_flag then
```

Two-Dimensional Tree Layout Algorithms

Initially some work was put into into producing a two-dimensional layout algorithm for computing the initial tree layouts. This immediately highlighted the complexity of such algorithms. Generally, attempts at producing an efficient version of such an algorithm proved mostly futile, and with the high availability of graph visualisation/layout libraries already available it quickly became clear that this was attempting to “reinvent the wheel” by manually coding the basic layout algorithm. For this reason, it was decided to simplify the initial layout process with the addition of a graph visualisation algorithm – specifically the Microsoft Automatic Graph Layout library (MSAGL – discussed in more detail in the Methodology / Software Libraries section of this document).

Ultimately to layout the tree in two dimensions my design became the following: parse a generic tree from a text file, adapt the tree to be MSAGL compliant, run MSAGL’s layout functions on that tree, then adapt the resulting MSAGL tree into Unity game objects to be rendered in 3D space. Because of the high dependency on a rigidly implemented exterior library for layout, and its minimal nature, pseudo code will not be provided for this algorithm.

Three-Dimensional Tree Layout Algorithms

To make basic use of the third dimension made available by the VR medium I designed a simple 3D transformation algorithm that takes a tree displayed in two dimensions and performs rotations along the axis of each branch to rotate the tree out of the plane. The algorithm takes as input a tree that is laid out on a plane using the “circular” view, then performs transformations upon it. This algorithm is quite straight forward in its implementation, the key concepts are shown in the following pseudocode implementation:

```
input  : A Tree object target_tree passed by reference
output : None - target_tree is mutated as a side effect

Function recursiveConstantRotate(current_node):
    degrees_per_child ← n /* 0 < n < 360 */
    foreach child_node in current_node do
        rotation_axis ← child_node positon - current_node position
        rotate child_node about rotation_axis by degrees_per_child
        recursiveConstantRotate (child_node)
    end
recursiveConstantRotate (root_node from target_tree)
```

Algorithm 4: Recursive Rotation

The above algorithm recursively visits each node in the tree (if called from the root) and rotates it about the axis between itself and its parent by a fixed quantity. This has the beneficial effect of reducing the branch crossings that can sometimes be generated by the MSAGL library’s layout methods. It should be noted that the pseudo code assumes that the final implementation honours the hierarchical nature of the tree and applies any rotation to both the current node and its children.

Terrain Generation

A key data view provided by this project is that of the terrain tree; this concept relies on utilising terrain height and shared terrain level to show node depth and node relationships respectively (see Fig 2 for an example of the final software implementation generating a tree). Generating the terrain took a considerable amount of computational effort and various versions of the generation algorithm. The process to create this algorithm will be explained in the following.

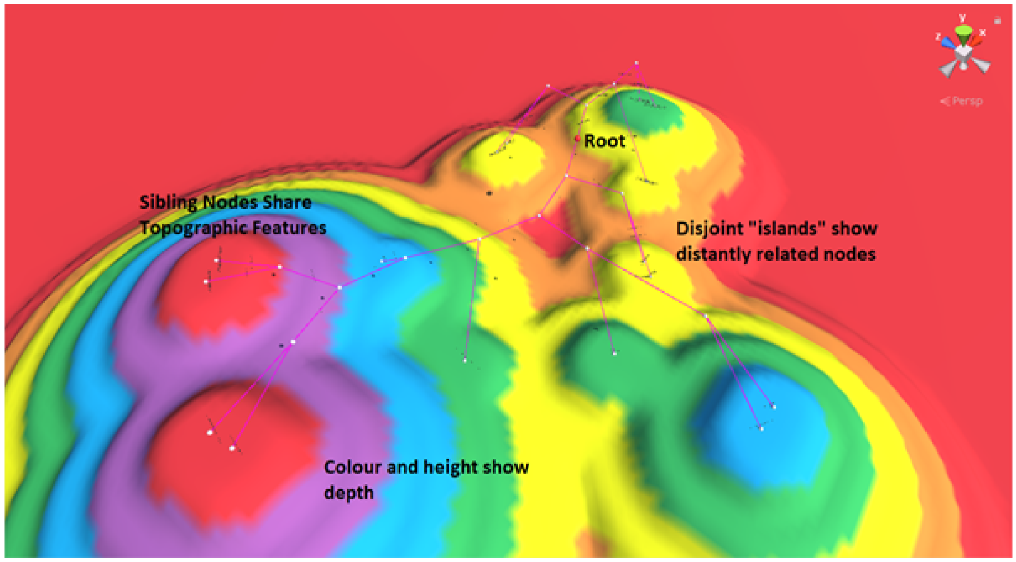

Fig 2. View of Terrain Tree. Generated from a Newick file.

To facilitate terrain generation programmatically, Unity utilises a height map feature. That is to say that all terrain in Unity is essentially encoded by a two-dimensional array of floating-point height values, representing the height of a discrete section of terrain at a coordinate specified by the indices in the heightmap array. So, to edit a section of terrain at runtime one simply has to change the values stored within the heightmap array of the terrain.

The colour of the terrain is actually maintained in a similar way; each texture assigned to the terrain has an "alphamap" related to it, which dictates how strongly the texture is applied at any point within the terrain.

Initially the algorithm simply took the force directed two-dimensional layout from the MSAGL library, rotated it 90 degrees about the Z axis to sit on the plane, and elevated a circle at each node within the terrain to a height dependent upon the node's depth. The result of this was unfortunately a terrain with little actual relevance to the tree data, and a generally poor representation of the tree itself.

This stage in development also highlighted the issue with nodes that were not siblings ending up too near each other in the final graph, resulting in terrains that spiked wildly in height from point to point, as a node of depth 10 might be a spatially close to a node of depth 6 for example. Similarly, it highlighted that iterating through the nodes in a depth first fashion, as we had, was a flawed decision.

Lastly during this step, it became clear that from a visual perspective some terrain smoothing would be necessary to make the generated features more visually pleasing.

To increase the information content of the generated terrain several steps were taken:

- Node traversal in breadth first order
- Consider only leaf nodes when raising terrain
- Application of a custom written force directed algorithm to bring sibling leaf nodes closer together and to repel non-sibling leaf nodes
- Application of a smoothing algorithm to counter jagged edges in the terrain

The first major step in developing suitable terrain was the application of a static force directed algorithm to group related nodes and repel non-related nodes, a concept that was inferred from the work of Waese and company [3]. While Waese et al's algorithm for force directed layout is applied at run time continuously "until a stable configuration is reached" [3], the algorithm employed in this case has to run a fixed number of times before the terrain is raised, as editing the terrain is too computationally expensive to be done at run time in the same way (due to the way in which heightmaps are stored and edited within Unity).

The developed algorithm therefore considers each leaf node in turn, then applies an attraction force to it and all sibling nodes that is equal to a force vector *Fbase* multiplied by the square of the distance between the two nodes dist. This means that at large ($dist > 1$) distances the two nodes are attracted together by a large force, while at smaller distances ($dist < 1$) the attraction force rapidly decreases. For every non-sibling node an opposite operation is performed: the two nodes are repelled from each other by -*Fbase* divided by the distance squared; this means that at small distances the repulsion force will be very large but will quickly diminish at greater distances.

The algorithm's pseudocode implementation is shown below:

```
input  : A list of leaf nodes leaf_nodes from a Tree object passed by reference
output : None - nodes are mutated as a side effect
foreach current_node in leaf_nodes do
    foreach other_nodes in leaf_nodes do
        dist_squared ← square of distance from current_node to other_nodes
        if current_node and other_nodes are siblings then
            if dist_squared > min distance then
                move current_node and other_nodes toward each other
            end
        else
            move current_node and other_nodes away from each other
        end
    end
end
```

Algorithm 5: Pass

The above algorithm is applied to the list of leaf nodes 40 times at tree generation time, resulting in tightly grouped sibling nodes and clear spatial separation between all other leaf nodes.

Smoothing a terrain heightmap is essentially an image processing problem, so to achieve my aims I implemented a simple mean box filter that uses averages to reduce harsh edges in the terrain. The algorithm essentially replaces the value of each heightmap point with the average value of its neighbours within a specified area. This algorithm is then applied a number of times in discrete passes to generate the final smoothed terrain. Ultimately it became clear that over smoothing the terrain had two problems – firstly having clear divisions between height levels was useful for data representation, and secondly that a high degree of smoothing was very computationally expensive to obtain. For this reason, I opted to utilise only a small number of passes and to average each point only with its immediate neighbours. The result of this smoothing can be seen again in figure 6. The pseudocode implementation for the inner smooth function "SmoothPass" is shown below:

```
input  : A 2D list (matrix) of heights (numbers) heights, a radius smooth_radius
output : A 2D list (matrix) of heights (numbers) new_heights

height_map_width ← x dimension of heights
height_map_height ← y dimension of heights
new_heights ← copy of heights

foreach column in heights do
    foreach row in heights do
        count ← 0
        total ← 0
        foreach value in heights within smooth_radius do
            count ← count + 1
            total ← total + value
        end
        new_heights at row, column ← total/ count
    end
end
```

Algorithm 6: Smooth Pass

At this point the terrain was both smooth and well grouped, but each node still appeared to sit on a "pillar" of terrain rather than in a fully layered topographic "landscape" as was the aim. To produce a more meaningful terrain two alterations to the generation process were required. Firstly, the terrain raising process was changed so that each node raised a set of concentric circles of increasing height and decreasing diameter – resulting in a "stack" of terrain appearing naturally beneath each node. Secondly, the method in which changes to the terrain occurred was altered by ensuring that an area of terrain's height would only be changed if the new height represented an increase to the terrain's height at that point. This ensured that the terrain would not be overridden or "clipped" by a disk of lower level being created in the same area as another disk of higher level (see Fig 3).

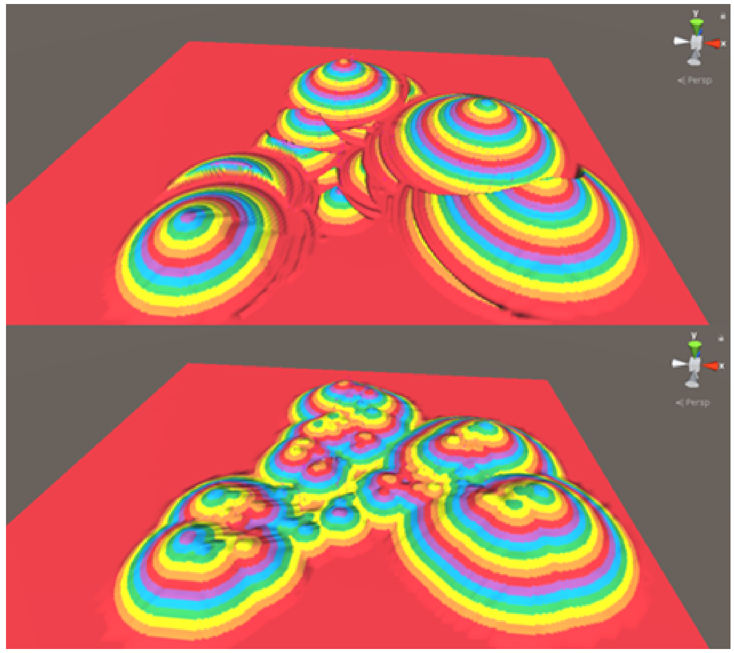

Fig 3. Height Clipping. Height overriding created by the lower disks of terrain generated for deeper nodes in the tree (top), vs the true terrain generated with a height comparison before assignment.

The pseudocode implementation of the final terrain raising algorithm is shown below:

```
input  : A list of leaf nodes BFS_ordered_leaf_nodes constructed breadth first
output : None - mutates heights contained within terrain as a side effect
/* <x, y> indicates a vector with components x and y   */
foreach current_node in BFS_ordered_leaf_nodes do
    height_increment ← 0.01
    current_depth ← 0
    min_size ← 20
    while current_depth < current_node depth do
        size ← (current_node depth - current_depth + 1) * min_size
        heights ← terrain height map around current_node
        radius_squared ← (size * size) / 4
        center ← < size/ 2, size/ 2] >
        for i in range 0 to size do
            for j in range 0 to size do
                index_pos ← < i, j >
                dist_squared ← square of distance between index_pos and center if
                  dist_squared < radius_squared then
                    val ← height_increment * current_depth
                    if val > heights [i, j] then
                        heights [i, j] ← val
                    end
                end
            end
        end
        update terrain heightmap at current_node with heights
        current_depth ← current_depth + 1
    end
end
```

Algorithm 7: Terrain Raising

Ultimately the above functions are combined in a specific order to produce the final terrain, along with the addition of caching functionality to write terrain heights to a text file – to reduce the time and computational requirement of generating larger trees.

The final high-level pseudo terrain generation function is as follows:

```
input  : None - runs on frame update if on_frame_loading_flag is true
output : None - mutates state of world objects as a side effect

num_raises_per_frame ← 10
current_build_step ← force_directed_layout
once_per_frame begin
    if on_frame_loading_flag then
        if current_build_step == force_directed_layout then
            perform force pass on leaf nodes of tree
            if number of passes performed >= number required then
                current_build_step ← check_for_cached_heights
            end
        else if current_build_step == check_for_cached_heights then
            if heights exist then
                load heights from cache
                current_build_step ← node_raising
            else
                current_build_step ← terrain_raising
            end
        else if current_build_step == terrain_raising then
            if bfs_queue_nodes is none then
                bfs_queue_nodes ← new queue
                enqueue root into bfs_queue_nodes
            end
            for i in range 0 to num_raises_per_frame do
                if bfs_queue_nodes is not empty then
                    dequeue current_node from bfs_queue_nodes
                    if current_node is leaf then
                        raise terrain to current_node
                    else
                        enqueue all children of current_node to bfs_queue_nodes
                    end
                else
                    current_build_step ← terrain_smoothing
                    break
                end
            end
        else if current_build_step == terrain_smoothing then
            smooth terrain
            cache terrain heights
            current_build_step ← node_raising
        else if current_build_step == node_raising then
            raise nodes above terrain
            raise user above terrain
            current_build_step ← terrain_colouring
        else if current_build_step == terrain_colouring then
            colour terrain based on new_heights
            on_frame_loading_flag ← false
            current_build_step ← force_directed_layout
    end
end
```

Algorithm 8: On Frame Loading

The function is designed to be run once per frame, each time completing a step or sub step in the full terrain generation process. This is done to ensure that the software doesn't become completely unresponsive during terrain building.

0.0.1 Metainformation and SQLite databases

An important element of this project was developing a sensible method of storing and retreiving meta information. Initially it had been suggested to perhaps support a different tree file standard with built in constructs for this data, but as the Newick format is so pervasive in current literature it made more sense to offer support for SQLite databases containing information on nodes.

To enable correct support for SQLite was unfortunately a non-trivial issue; Android has native support for SQLite databases (this is ultimately the target build platform for the project), but Unity unfortunately offers no such support. Therefore, to enable the utilization of databases a plugin developed under the MIT permissive license was used, developed by Asif R and made available through GitHub. The article explaining this plugin's use and linking to the GitHub repository is available via [6]. With this plugin included in the Unity build a script DBQueries could be written to facilitate the querying of a database. The aim of this inclusion was to allow the user to provide a separate database file with the same name as a given tree but the ".db" file extension. This is then loaded by my software and checked for a "metainformation" table; the table is expected to be indexed by a primary key column labelled "id". If the table matches the expected format column names are extracted and stored, and then using standard SQLite queries (via Asif R's SQLite plugin 2018 [6]) the table is queried for information with relation to each node in the tree using a simple SELECT/FROM command filtered by ID. Each node's internal string variable "metainformation" is then updated to be a concatenation of all displayable data from the tree tagged with its column name.

Additional unity scripts are then used at run time to display the metainformation to the user when they hover over a node. An example meta information panel is shown in Fig 4

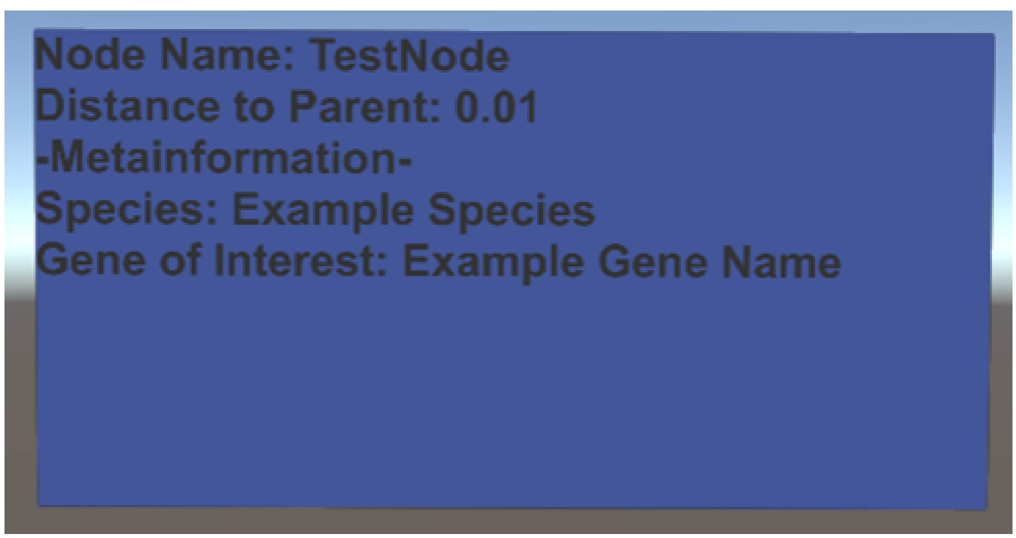

Fig 4. Example metainformation panel. Generated from associated SQLite database.

Results

The developed software is capable of representing small to medium / large sized trees efficiently, with a high degree of accuracy in terms of the terrain layouts generated. It takes around a minute to fully generate the terrain for larger trees, but this is then cached to the device storage so that the computation doesn't have to be repeated. The other views provided all generate rapidly, and there is discernable lag from the user's perspective.

By utilising each of the data views provided, a user can easily glean key information about a tree of interest. The user can load the tree in the terrain view to understand high level information, then "zoom in" or switch to one of the other more basic tree views to understand specific areas of interest. This is a unique way of understanding tree data that - to the author's knowledge - has not as yet been replicated. There is a high degree of potential for both this software and any others that are capable of representing trees in a multitude of ways to maximise the efficiency with which researchers can understand them.

Discussion

The software developed for this project proved to be a novel and unique first step into the possibilities of representing phylogenetic information in VR. Both the algorithms and the final application serve as an excellent indicator of the potential of using VR for viewing phylogenetic data along with applying to the area of data visualisation as a whole. The data views developed provide an excellent accompaniment to the standard tree diagram - by both extending it with supplementary meta information, and by altering it altogether and displaying it as a terrain based view.

From the current state of the project there are a great number of potential improvements and additions one could make to the software. While in conversation with an end user a good number of these suggestions were brought up. Namely it became apparent that while this project represented a good start in the realm of displaying phylogenetic trees, there was a great deal of potential for other information visualisation methods to be developed.

Firstly, in terms of concrete improvements, the terrain view implemented would benefit greatly from including a better representation of phylogenetic distance (as suggested during user testing). The terrain layout algorithm could be altered to achieve this without too much difficulty, although it would be necessary to manufacture the algorithm to ensure that the application does not become unresponsive.

Secondly, a more consistent and clear way of managing tree files would be helpful; perhaps the inclusion of an online service to transfer files to the device could be of use, allowing the user to simply upload their newick files to a website and then access them through the app "on device". This could also perhaps apply to tree terrain generation also – the terrain could be generated using a cloud computing service and the resulting heightmap passed cached on to the device instead, reducing the amount of computation that need be done on the comparatively slow headset processor.

Thirdly, it could be prudent to put some time into researching ways of allowing users to see both high- and low-level features of trees – and to give them fluid methods of transitioning between these views. Perhaps this could take the form of a terrain that represented the high-level features and spacing of nodes, with the view transitioning into a clearer branching local structure as the user zoomed in? There is a great deal of potential in this area, and a full design team working in contact with actual researchers in the field could make great strides in improving the usability and usefulness of VR visualisation systems. Fourthly, there is an opening for a system that is able to show the different possible trees that can be created from the same species; given a group of species one can construct an evolutionary tree between them based on the inheritance of one gene that might be wildly different from another tree generated based on the inheritance of a separate gene. A system that could show the variations in these trees and highlight how the models vary based on the supplied information could be a fantastic tool for researchers.

Fifth, the importance of efficiency must be considered. As pointed out previously there is a vast amount of data available, and any tool capable of representing modern trees with their potentially thousands if not millions of leaf nodes is likely to be invaluable to researchers.

Ultimately through working on this project it has become clear that new and better visualisation tools are needed, and in whatever medium they take they have to meet one key requirement: They must be "better than just using a bigger piece of paper" (as pointed out by Dr K. Winzer during testing). Whatever systems are developed must represent an actual benefit for the target audience vs simply printing a cladogram out and annotating it by hand – there has to be a proper translation from the academic task of designing the systems to the actual implementation and usefulness of said systems. The system we developed proves that visualising phylogenetic trees in VR is both possible and an area worthy of further study but ultimately could be extended and improved immensely with more time and more developers.

Acknowledgments

We extend thanks to Dr K. Winzer for discussing the final application with us and giving useful feedback on potential future improvements.

References

1. Emery L. Why is pylogenetics important? *European Bioinformatics Institute [Internet]*. 2014 May [cited 2018 May 05];9(12):[about 1p.]. Available from: https://www.ebi.ac.uk/training/online/course/introduction-phylogenetics/why-phylogenetics-important

2. Carrizo SF. Phylogenetic Trees: An Information Visualisation Perspective. *Proceedings of the Second Conference on Asia-Pacific Bioinformatics - Volume 29*. 2004 Jan; 29: 315–320 Available from: https://dl.acm.org/citation.cfm?id=976520.976563 [Accessed 2019 June 16]

3. Waese J, Provart NJ, Guttman DS. Topo-phylogeny: Visualizing evolutionary relationships on a topographic landscape. *PLoS ONE*. 2017 May; 12(5): e0175895. Available from: https://doi.org/10.1371/journal.pone.0175895

4. Forghani M, Vasev P, Averbukh V. A Visualization System for Binary Rooted Trees. *Scientific Visualisation*. 2017 Dec; 9(4):59-66. Available from: http://sv-journal.org/2017-4/06/index.php?lang=en doi: 10.26583/sv.9.4.06.

5. Craighead J, Burke J, Murphy R. Using the unity game engine to develop sarge: a case study. *Computer* 2007 Jan [cited 2019 Jun 18]; 4552. Available from: https://www.researchgate.net/publication/265284198_Using_the_Unity_Game_Engine_to_D

6. Asif R *SQLite and unity: how to do it right.* Available from: https://medium.com/@rizasif92/sqlite-and-unity-how-to-do-it-right-31991712190 [Accessed 2019 Jun 20]