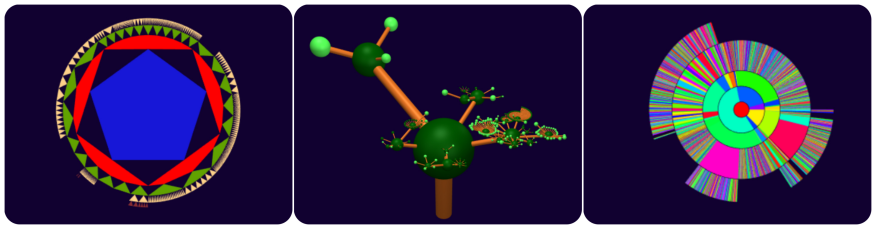
VizWick: A Multiperspective View of Hierarchical Data

Adrian Vrămuleţ, Alex Thieme, Alina Vorobiova, Denis Shehu,
Mara Miulescu, Mehrdad Farsadyar, Tar van Krieken



Abstract—In this paper we present a web-based interactive tool for visualizing hierarchical data. Our main purpose is to facilitate the visualization of data sets for everyone, from anywhere in the world. We achieve this by offering VizWick in a browser environment, with no requirement of additional software. Furthermore, the web page has an intuitive design, making it easy for the user to become accustomed to it and to utilize the tool. Our tool provides the option to view the same data set from multiple coordinated perspectives, thus giving the user the possibility to gain more analytical insight than if the data set were visualized in single view. There is a range of visualization techniques to choose from, which can be either in 2D, 3D, or Virtual Reality environment. The choice of programming language for this project is JavaScript, with the aid of PixiJS and Three.js libraries. We demonstrate the working of our tool using the NCBI taxonomy, an hierarchically structured data set which contains over 300 000 items.

Index Terms—Web-based interaction, Hierarchical data, Multiple coordinated perspectives.

1 Introduction

In this paper we describe VizWick, a web-based interactive tool for visualizing hierarchical data. It offers both ease of access and ease of use – we have built it in a browser environment using JavaScript, PixiJS and Three.js, and we have designed it such that people from the industry and laypeople can use the tool just as easily as one another, by means of a simple data uploading button and familiar interactions such as select, expand, collapse. Furthermore, users can explore statistics about their data set by means of a sidebar showing information such as the tree depth and the descendant count.

This tool gives the user the possibility to visualize their data set from multiple perspectives, among which we mention customized implementations of Froth, Self-Adapting Sunburst, and Diamond Tree, as well as the new visualizations Acacia Tree and Jesterhat - all of these techniques are supported in 2D space, apart from Acacia Tree which is offered in 3D and has support for viewing in VR. The visualizations can be viewed in different formats - one, two, or four perspectives on the screen simultaneously, where an interaction with one visualization triggers the same interaction for all other visualizations currently in view.

VizWick takes as input hierarchically structured data sets in the Newick format - hence its name. We will illustrate the utility of the program by means of the NCBI taxonomy data set, which contains more than 300,000 species organized hierarchically, and which adheres to the Newick standard.

2 Related Work

There exists a wide range of tree visualization methods to encode hierarchical data. Each of these methods have their pros and cons depending on the particular tasks for which they are used. In this section we review some of these visualization methods and evaluate their advantages and disadvantages.

Self Adapting Sunburst (SAS) is a space-filling technique with a layout inspired from Sunburst visualization. However, compared to Sunburst, SAS allocates area to each node based on the node's attribute value, and arranges the children of any node in the order of their area size. Here, nodes that are in the same hierarchical level occupy the same ring in the visualization, and the width of the circular rings is set dynamically based on the number of nodes in that layer. It is argued that these features will allow SAS to achieve a better space utilization rate than Sunburst. However, because SAS shows only a limited number of layers of hierarchy on the screen at a time, it has limited usefulness for visualizing large data sets with many layers (deep hierarchies) [1].

Fig. 1: Self Adapting Sunburst

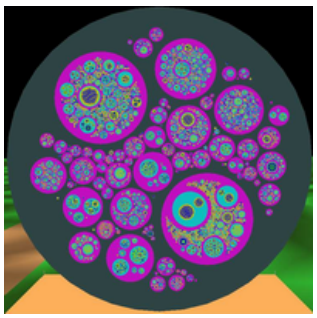


Fig. 2: Froth

Froth is defined as the implementation of a force-directed tree layout algorithm based on the Lennard-Jones potential. This technique visualizes hierarchical data using circles, namely it follows the containment convention of drawing trees, where subtrees are displayed as small disks contained in their parent-node disk. Underlying Froth is a divide-and-conquer algorithm which makes the visualization fast, and a force-directed simulation process which ensures smooth animation. In its original implementation made by its creators, Froth comes with a series of advantages - it supports dynamic insertion and deletion while maintaining the running time, and also features efficient space filling and uniform edge length because of its analogy to a physical system [2].

For our implementation, however, Froth's efficient space filling is a disadvantage as it is a complex and difficult mathematical task which we may not fulfill.

Orthographic Treemap is a kind of visualization which uses orthographic projection and animation to convey Treemap structure. The main idea is to create a series of planar layers comprised of tiles generated from each node in the source hierarchy. The resultant layers are arranged into an orthographic stack with the root tile at the top. The main benefit of this arrangement is that the levels can be easily identified and labeled. Interaction and animation help user to compare nodes and understand the structure. However, the accommodating hierarchies where leaf nodes are not necessary on the same depth make this visualization hard to implement. Also, when we have a really big hierarchy tree the sizes of tiles will be too small for comfortable user interaction [3].

Fig. 3: Orthographic Treemap

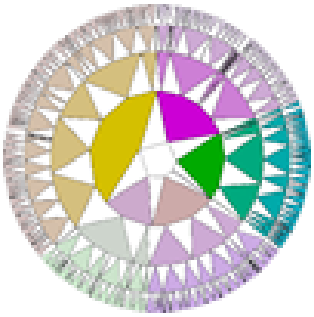


Fig. 4: Diamond Tree

Diamond Tree is a 2D visualization which aims to make the best use of screen estate. To arrive at an impressive diamond tree a process of three stages lies ahead. Firstly, a general - not detailed - design of the tree is generated where in its center always a polygon is situated. The latter one's number of sides is equal to the number of children of the root. Secondly, the existing design is detailed further by changing the sizes of each node according to the size of the subtree rooted at it. Finally, the visualization is optimized continuously until it efficiently uses the given space. In this visualization all the nodes except for the root appear as arcs whose distances from the latter are equal if they are in the same level. A data set which has less than 5000 data points suits the diamond tree best (if desired to be fully shown in the screen), but even if this is satisfied, if the tree is not somehow balanced then this visualization is not the one you should go for [4].

Career Tree is a visualization specifically created to visualize someone's career. To do this, it pulls data from a user's LinkedIn profile. The lower branches in the tree represent the user's education and the higher branches contain nodes that represent the user's professional titles. It is a very basic visualization which was created in a very specific environment for a specific set of data. The basic principle is just a simple hierarchical tree structure however, just made to also look like an actual tree that can be found in nature. The main appeal of this visualization is simply its appearance, it is a nice simple yet pleasing diagram. It is not a very efficient visualization however, as there is a lot of empty space in the visualization, and it is rather chaotic by design [5].


Fig. 5: Career Tree

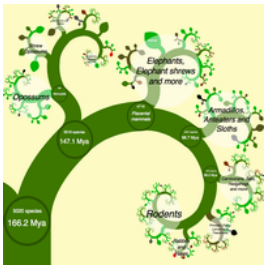


Fig. 6: One Zoom

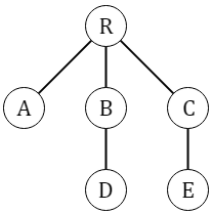
One Zoom is an interactive technique quite similar to the zoom function in Google Maps, specifically designed to allow for looking at deep trees. It allows for a theoretically infinite zooming system, recursively rendering more details as the zoom increases. This technique can be applied to most visualizations, though it might reduce the amount of overview a visualization can grant, at the benefit of introducing more detail [6].

Pythagoras Tree is a fractal that can be used to visualize hierarchies. Its name comes from its construction, namely because every new branch creates a right triangle. Firstly, a square of length c is drawn. Then, two squares are attached to one of its sides, usually on the top one. Their size depends on a weight function in terms of information contained only in further directly linked subbranches. The squares represent vertices in the hierarchy. Eventually, the two acute angles in every right triangle determine the curvature of the whole tree, depending on their values. Then, this process is repeated on the new squares up to a predetermined depth. One of this technique's advantages is that the visualization is rather intuitive, which allows for easier implementation and further customization [7].


Fig. 7: Pythagoras Tree

3 Data Model

For this project we are working with hierarchical data sets, which as the name suggests are data models in which data is organized in a tree-like structure - an example of that is any simple family tree or the file structure on our personal computers [8]. Our project is compatible with only those trees which are represented by the Newick format where only the nodes are named - and the distances are not present. Let us explain this by giving an example: suppose we have the tree shown in the figure to the right. Then by using the above Newick format its representation would be the following: (A, (D)B, (E)C)R;.



The advantage of using this Newick format is that it conveys as much information for each node by using as little characters as possible, and conciseness is very important when you have a data set with over 300 000 data points. However, this conciseness has its drawbacks because no information about the distances are given in the above Newick format and this might restrict the number of users that would like to use our tool.

Once the file is uploaded, its textual content is stored as a string in a field and then it is iterated character by character. During this process all the nodes are created and their relation to their parent is stored in a field. The final outcome of the whole process is an object composed of nodes within which there are other nodes and so on. Each node has fields that store information like the parent, an array of its children, etc. This object is then used by other sections of the code in order to create a data structure that represents a hierarchical data tree [9].

4 The Visualization Tool

4.1 Graphical User Interface

The web page of VizWick consists of three distinct pages, namely: the home page, the preview page, and the visualization page. But why did

we choose these three?

Originally, we thought that two would be enough, those were: a page that displayed a welcome message (this was an early version of the home page, see Fig. 9) and a visualization page as can be seen in the final version of the web page (see Fig.). Both these pages underwent major transformations and several elements were added, removed, or altered – which we will come back to later – and a third page came into existence (i.e. the preview page). The function of the latter one is to inform those users who are not familiar with our web page, by giving a short description of VizWick and its five visualizations – all of them come with a respective photo.

Before we started with the actual development of the web page we came up with a simple design on paper (no colors, font styles, or font sizes), then we 'translated' that to an HTML file – which has an external link to a CSS file. Along the way the actual design of the web page started to differ from the one on paper because we had to deal with complications and changes that we were not able to foresee at the moment of drawing the initial design. However, we now have a fully functioning web page whose structure is explained below in detail.

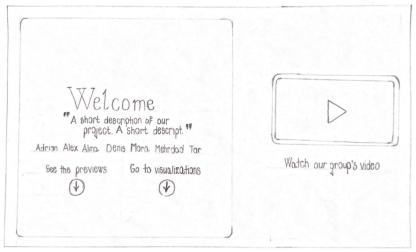


Fig. 8: The initial home page

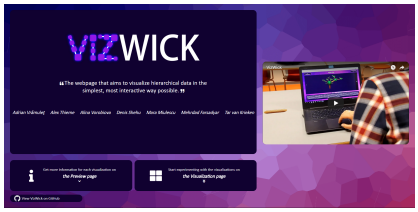


Fig. 9: The current home page

Firstly, let us have a look at **the home page**. As the name suggests it is the first page that the user sees when (s)he clicks on VizWick's link – i.e. `https://aaal-e.github.io/VizWick/` – and it is composed of two sections, namely the left and the right sections. The first-mentioned one is also divided into two subparts: the upper and the lower ones. The upper subpart contains VizWick's logo, a short description of less than three lines of our project, and our group member's names. The lower subpart has three useful links: two internal HTML links that direct the user to the other two pages of the web page and one external link to VizWick's GitHub page where the user has the opportunity to view our entire code (because as stated previously our project is open source).

Whereas the right side of the screen is less crowded than the previous one and only contains our group's presentation video which in seven minutes explains the aim, usage and the building process of our web page. As you may conclude, the home page just gives a short introduction to our project and team and thus it is not a user-interactable page. In order to do that you have to visit the visualization page.

But before that let us analyze the second page of VizWick, i.e. **the preview page**.It contains just a slideshow which covers the whole screen estate and it has six slides: the first one being about our project in general and the other five about our five visualizations. The slideshow area is composed of three parts: the title of the current slide is shown at the top, an image that displays the object we are referring to occupies the middle section, and a description about it at the bottom end of the page.

Fig. 10: The preview page

Furthermore, to the left and the right sides of the slide's title there are two navigation button which will direct you to the home page (in any case if the user wishes to go back to the start of the web page) and to the visualization page respectively. Also at the left and the right sides of the slide's image we have two other HTML elements, but here they are just two simple arrows that help the uses to navigate between the six different slides of the slideshow. By clicking on the arrow on the left the user will go to the previous slide and similarly when clicking on the arrow on the right (s)he will go to the next slide. Another way to navigate between slides – and to find out which one is currently shown in the screen – is by the six dots located between the image and the description of the slideshow (each of them corresponds to one of the six slides). They have a dark bluish color by default, but when clicked they change to white, thus the dot of the slide displayed on the screen will always be white – this way the user can see in which slide (s)he currently is.

The last part of the slideshow is the description at the bottom whose text is taken from each visualization's JavaScript file; because when we created our visualizations, we specified an image and description variable on their code and these two are taken by the web page and displayed on the slideshow, one as the image and one as the description of the visualization.

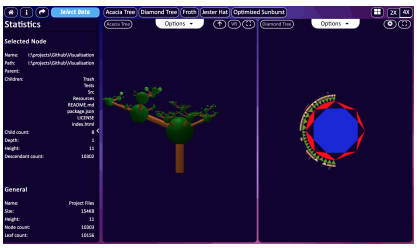


Fig. 11: The visualization page, two layout

Lastly, let us dive into the structure of the most sophisticated page of VizWick, the one where everything happens: **the visualization page**. This page consists of the top section (which is a very thin layer of buttons at the top of the screen), the left section (a collapsible statistics window), and the right section (the part of the screen where the visualizations will be actually displayed). When the user opens the visualization page for the first time (s)he will find a pop-up window at the middle of the screen which requests him/her to upload the data set. This window is divided into two parts: the left one shows the two already existing data sets (the NCBI Taxonomy and our project files data set) that the user can choose to visualize if (s)he does not have a data set yet and the right part which is a drag and drop area where the user can drop his/her data set file in order to upload it to VizWick. After the file is uploaded a pop-up message will show up at the top indicating that the file is loaded successfully.

The top section, starting from the left to the right, contains the following buttons: the two navigation buttons that direct the user to the home page and the preview page respectively, a share button that generates a link which the user can share with anyone (it preserves the selected visualizations and the position down the tree at the moment it is generated, but we will talk later about it in more detail), the 'Select Data' button which gives rise to the pop-up window mention above, then each of the visualizations have a 'button' like element with which you can use drag and drop (by dropping it in any of the visualization areas the user will be able to display the respective visualization in that particular area). At the far left there is a reset button which by what the name indicates resets the sizes of the visualization areas – because these latter areas are resizable both horizontally and vertically, but by default they have the same dimensions – and there are also the 'two layout'/2X and the 'four layout'/4X buttons which divide the visualization area into respectively two and four distinct areas, so the user can have two or four different visualizations displayed at the same

time.

The share button when clicked will cause a pop-up window to appear at the top of the screen asking the user if (s)he wishes to get a sharable link. If the user clicks the button 'continue' then the link is generated and another pop-up window appears asking the user if (s)he wishes to copy it. When that link is being followed the user will automatically go to the visualization page, a pop-up message is displayed saying that the data set is being loaded, and after a bit the visualizations will appear as they were when the link was being generated. The statistics window of the visualization page contains useful information about the data set in general and about the currently selected node – if there is any – examples of these are: the height of the tree, the total number of nodes/leaves, the name of the selected node, its parent, a list of its children, etc. This window is collapsible and you can achieve that by clicking on the arrow at the rightmost part of it.

Then we have the visualization area which can be composed of two or four distinct areas where any visualization can be visualized. Every area has a non-usable section at the top or bottom – depending on the position of the visualization area in the 'four layout' version (if it is at the top then this section is also at the top and similarly if it is at the bottom the non-usable section is at the bottom) – which is relatively small compared to the whole visualization area and it contains the text 'Options' with an arrow at the right pointing upward/downwards (it strongly resembles the shape of the notch on the iPhone X). When this notch is clicked an option pane slides down/up and if there is a visualization at this particular visualization area then it will contain several options (if not it will display 'No options are available'). At the left of the notch there is nothing when no visualization is visualized, but when there is its name appears and on the right there is always a full screen button and other buttons depending on the visualization (e.g. VR button, 'View the whole visualization' button, etc.). When a visualization area is in full screen the top section part of the page is removed and if the user also collapses the statistics window then (s)he will truly have a full screen visualization of his/her data set.

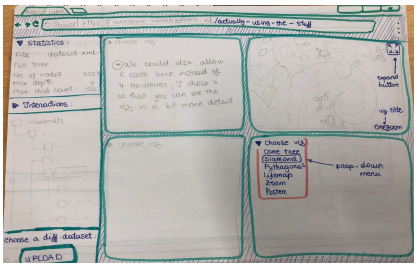


Fig. 12: The initial 'four layout'

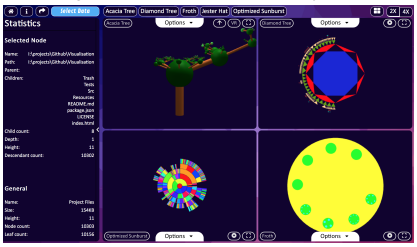


Fig. 13: The current 'four layout'

We chose to have a 'two layout'/2X and a 'four layout'/4X because in some occasions – e.g. when the data set is complex – the user might need four different visualizations side by side in order to grasp the hierarchical structure and in certain occasions (s)he just needs at most two – by using the 'two layout' version the screen is not that cluttered. Along these lines we placed the option's notch at the bottom for the two visualization areas at the bottom of the screen at the 'four layout' version (and not at the top as the other two) because we did not want the user to get distracted when comparing the different visualizations. The reason why we choose the visualization areas to be sizeable is because the user might sometimes want to focus on the details of a particular visualizations and on the bulk structure of another. And lastly, we chose to have a 'Share' button instead of an open explorable database of data sets because we expect many users to be working in teams and thus wanting an easy and fast way of sharing data with colleagues, which they can achieve by using this button.

4.2 Visualization Techniques

In this section we discuss our chosen 2D visualizations in detail, followed by a description of our 3D visualization ideas.

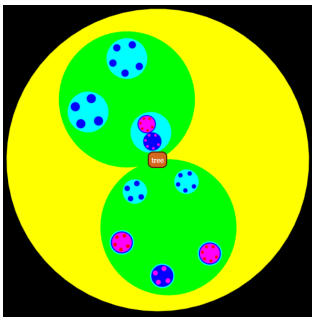


Fig. 14: Froth

Froth: We liked the idea of Froth, but Froth has a problematic aspect associated with it, namely circle packing. Circle packing is a mathematical problem, of how to best fill a circle with other circles, in this case of set ratios between the smaller circles. This problem requires complex geometry to solve on a case by case basis if one wants to have the optimal solution. Since we did want to implement a visualization similar to Froth, but did not want to have to spend too many resources on trying to optimize circle packing, we have used an algorithm that leaves quite some empty space, but looks intuitive and is useful. Furthermore, we implemented an interaction based on OneZoom, allowing us to recursively render the tree as one zooms in, only showing the parts that are on screen. It is also possible to move the screen around, dragging and dropping it in a way also resembling Google maps. this makes using this visualization quite intuitive, since most people are already familiar with these controls.

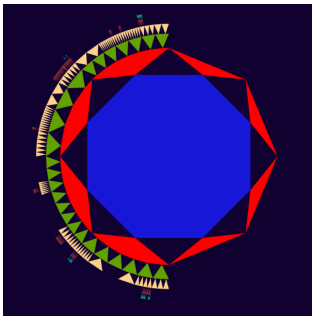


Fig. 15: Diamond Tree

Diamond Tree: We kept the default visualization technique for the Diamond Tree, however we made some minor changes in the process of constructing the tree.

It starts by placing the root of the tree in the center of the screen; it is displayed as a convex polygon - not necessarily regular because its edges' length depend on the size of the subtree root at its children. Then all the other nodes are displayed and their shape is not a perfect arc, but a mixture between the latter one and a triangle. The nodes that are in the same level in the tree will appear in the same ring in the diamond tree and will have the same radius, but their circumstances will differ - according to the size of the subtree rooted at them. These level-rings are not complete because there exists some space between each arc and their radii differ from ring to ring, arcs that are closer to the root have a larger radius, then those that are further away and between two consecutive rings have their radii increased by 30-40%.

Because of the large size of data the whole Diamond Tree cannot be displayed in just one screen. To counter this we had to implement interactions like dynamic zooming, expanding and collapsing of the nodes, resizing of the whole visualization.

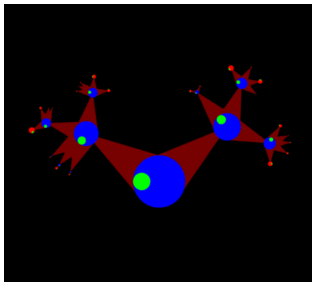


Fig. 16: Jesterhat

JesterHat: We have designed a visualization based loosely on the idea of a Pythagoras Tree. It assigns every node with a set angle space, where it can grant its children parts according to the size of their subtree. This happens recursively, creating a structure similar to a Pythagoras Tree, but stylised to look similar to a jester's hat. This visualization was made with both keyboard controls as well as mouse controls in mind, allowing one to walk through the tree with nothing but the arrow keys, or click their way around at their pleasure. Each method of control allows the same amount of options and functionality.

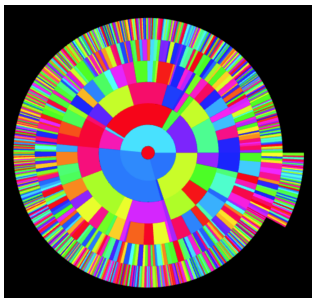


Fig. 17: Optimized Sunburst

Optimized Sunburst (OS): We designed this visualization by combining the best features of SAS and FanLens visualization techniques, and then adding extra features to make it more readable and efficient in use of space.
The general working mechanism, similar to Sunburst, is based on a radial layout, with the root located at the center and the nodes on the same level occupying the same ring. Similar to SAS, the thickness of a ring is dynamically determined by the number of nodes in that level.
The allocated area size of a node on each ring is proportional to the subtree-size rooted at that node; however, instead of the complete subtree we consider only the first X levels of that subtree (where X is a fixed number or set dynamically). The advantage of this is that the nodes with more visible children are assigned larger areas.

```
// each node calculates thickness for its child layer
calcChildLayer_Thickness(){
    // standard minimum thickness:
    var childLayer_Thickness = 60;

    // layerHeadCountArray holds the number of nodes
    // in each layer:
    var childLayerPopulation =
                    layerHeadCountArray[this.layerNumber + 1];

    // calculate the circumference of the child layer
    // and divide it by the number of nodes in that
    // layer to determine how dense that layer is
    var avgPixelPerNode =
            this.radialBand.getOutRadius()*2*Math.PI
                / childLayerPopulation;

    // assign thickness based on how dense the layer
    // is, if not dense then use standard thickness
    if (avgPixelPerNode < 2){
            childLayer_Thickness = 120;
    } else if (avgPixelPerNode < 8) {
            childLayer_Thickness = 90;
    }
    return childLayer_Thickness;
}
```

The following interactions are planned: 1) hover over a node to see its description, 2) click on any node and the subtree rooted at that node is visualized by putting that node at the center of the visualization.

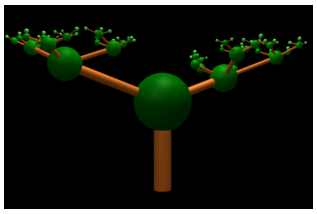


Fig. 18: Acacia Tree

Acacia Tree: Acacia tree was designed to represent an actual tree in nature. It does this in a rather simple way. Every node is represented by a green sphere in 3D space and has a brown cylinder connecting to its parent. All nodes are laid out in a circle above their parent, and decrease in size relative to their parent. The main inspiration for this type of visualization came from the Career Tree [5].
One of the features of this visualization is that the scale of the children is not linearly dependent on the fraction of nodes that their subtree has of the parent's subtree. Because in many occasions, this would result in almost invisible nodes, as they would have become tiny. Instead, the Acacia Tree opts for having them be dependent according to a logarithm, with in our case approximately base 33. This base was arbitrary chosen to look decent with our test data sets, as it is very difficult to reason about a proper number.

```
/*define some settings on how to scale*/
//the default ratio between parent and child size
const constantDecrease = 2;
//how important it is to scale according to the subtree size
const constantTreeSizeSignificance = 0.03;
const logbase = log(1/constantTreeSizeSignificance);

/*calculate the scale modifier based on subtree size*/
var pstnc = parent.getSubTreeNodeCount()-1;
var treeSizeFraction = pstnc/child.getSubTreeNodeCount();
var treeSizeScaler = 1 - log(treeSizeFraction) / logbase;

/*apply the scale*/
child.scale = parent.scale/constantDecrease/treeSizeScaler;
```

An idea that could be worth exploring but has not been explored by us, is analyzing the data set up front and check the subtree ratios to base an optimal logarithm on that.
The other core feature of this visualization is that it makes use of 3D space. By doing so, it is much less likely that nodes will overlap. We are however only seeing a 2D projection of this 3D space, so visually these nodes will instead happen to overlap more often. Humans are however very good at extracting this depth data from 2D projections, so this data is not completely lost. Nonetheless, this makes the visualization a bit worse when only displaying a static image, as the overview will be more chaotic. But when making use of the interactive tool, the user is able to rotate the camera in 3D space in order to look at the relevant parts from the correct perspective. It can however still happen that the shapes intersect in 3D space, which is why the user can manually drag shapes around by right clicking in order to resolve this issue.
This visualization mainly becomes very effective when the user owns a VR system, as our framework allows any 3D visualization to be viewed in VR. When using VR, this depth data is not lost at all like it is on your monitor. Each eye receives an image from a slightly different perspective, from which our brains can accurately extract the depth information. Therefore, when using this visualization in VR, it really is like it exists in 3D space. So now, only when a node is entirely hidden behind another node, we have a slight problem. The user can however easily move his/her head a bit sideways in order to change perspective. Moreover, when using spatially tracked controllers, the user can now interact with this visualization in a very intuitive way. The user is for instance able to move, scale and rotate the visualization as a whole And any interactions that the visualization had to offer when using the mouse, can now be executed using the controller. One can simply point this the controller at a node, and press the trigger.

4.3 Implementation Details

For this tool we decided not to use any existing visualization framework. Instead we created our own framework, such that we have complete control over the system and do not depend on what is implemented in some other framework. We do still depend on some libraries for rendering, but these could be swapped out for others if they do not serve our needs. All their data is wrapped in our classes such that we wouldn't have to refactor each individual visualization when this happens. This allows our tool to be rather well extendable and future proof.

This framework consists of 3 main parts, the core system that tracks and connects all the data, the 2D, and 3D wrappers that use libraries in order to render this data.

The core system is made in such a way that it will only attempt to show a limited number of nodes at once, and allow visualizations to easily dynamically load other data that the user wants to look at. So our system never attempts to visualize the entire data set, but only loads a subset. Each visualization will make sure that it actually shows the data that the user wants to see by adding intuitive interactions. This core system consists of 5 parts of its own. We firstly have the Graphics class which tracks all the shapes that are currently visible in the screen. It also has an event system such that events like clicks in the visualization can be detected. The next class that we have is the shape class, this will contain data such as the location on the screen, the rotation of the shape and the scale of the shape. It always takes the graphics as an argument, such that it is connected to the appropriate graphics right away, but it isn't rendered until we explicitly tell it to make itself visible. Every graphics instance also contains an instance of the Camera class, this camera will have a location, rotation and scale, and will decide from what location we view the visualization. These classes are all general classes for the rendering however, none of it is specific the visualization system yet. For that we have 2 extra classes, the NodeShape class and Visualization class. The NodeShape class extends the Shape class, such that it also has a location and these other common attributes, but it adds hierarchical and node related methods. Every NodeShape instance will have a reference to the actual data node in order to extract stats from this to be visualized, and it adds methods like getChildren in order to get the child NodeShapes of the NodeShape. The Visualization class extends the graphics class, but adds some things to it like keeping track of the NodeShapes that are displayed and synchronization methods that can be used to synchronize between different visualizations.

The general shape class also has some nice features that can be used to make the visualization more interactive and animated. It for instance contains a system that stores the shapes in a spatial tree in order to retrieve shapes that are close to one and another in an efficient manner. This can then be used to find nearby shapes and apply physics to push these shapes apart such that they don't overlap. Shapes also contain velocity data which will be used to change the location, rotation and scale continuously if not equal to zero. These methods greatly simplify making the visualizations more animated.

The 2D and 3D systems will extend these core classes and override methods where necessary in order to connect the data of their respective library to the class. This means that all the methods that are available for the 2D and 3D systems are nearly identical. The 2D system will however ignore most of the 3D related data, such as x- and y-rotation, because we cannot display this in a 2 dimensional context. For the 2D context we also define basic shapes such as squares, circles and polygons. And similarly we define basic shapes such as cuboids, spheres and lights for the 3D context. Both contexts also have some special shapes however, such as shape groups that can hold a collection of child shapes that will be transformed according to this shape. So if this shape groups scales down, so will all its children. The NodeShape will in fact extend this ShapeGroup class as a NodeShape will be represented by the child shapes that are added to it. Both contexts also have a special HTML shape available that will render a shape as an HTML element on the screen, but synchronizes the location of this HTML element with the location of the shape in the visualization.

The libraries that we wrap our system around are PixiJS and Three.js for 2D and 3D contexts respectively. PixiJS is a library that is rather popular for 2D rendering on the web. It uses of WebGL in order to have high performance by making use of the GPU as much as possible. Three.js is a library similar to PixiJS, but aimed at 3dDrendering. This library also is rather popular, and makes use of WebGL. One of the main reasons for choosing for this library was also the fact that it has WebVR support. We have made use of this in our platform, such that it now allows users to view any of the 3D visualizations in VR. Unfortunately, at the date of writing this, only Firefox has support for WebVR. So other browsers will just show a message telling the user to use Firefox if they want to make use of the WebVR feature.

The visualization and NodeShape classes also depend on some other systems to be in place. One of these is the actual tree containing the data to be visualized. We have a function in place to read the Newick format and convert this into an plain hierarchical structure that JavaScript understands. This object then gets passed to our Tree class, which will wrap its data into node objects that will compute standard statistics about the nodes such as their descendant count, as well as add some methods to retrieve any relations. This node also has 1 more very important task, it keeps track of the NodeShapes that are currently representing the node in different visualizations. This is then used whenever hovering over a node in order to synchronize the states of these NodeShapes in constant time. Another class that the visualizations depend on is the VisualizationHandler class. This class ties together all the components of the tool, it will keep track of all the areas that are able to display visualizations, all the available visualization types, the nodes that are currently focused and highlighted and the data Tree that is loaded. Whenever we want to change what visualization is shown in a certain area, we just tell the visualizationHandler to show the visualization with a certain name in the area with a certain name, and it will take care of everything else. A slightly simplified model of the web page can be seen in the following figure.

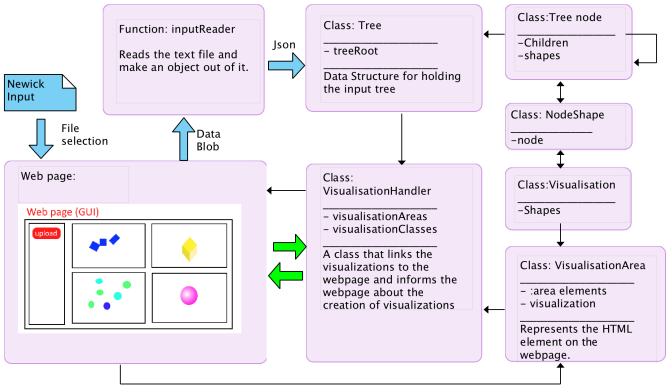


Fig. 19: UML diagram showing the page structure

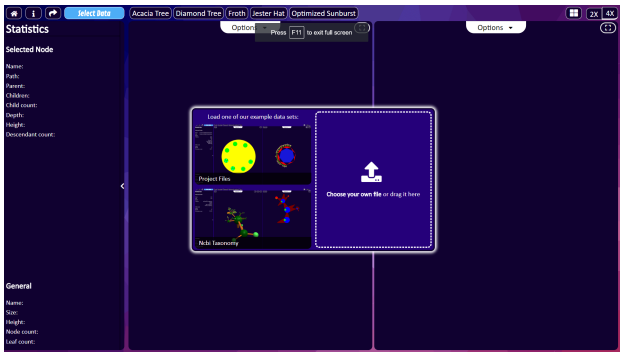
Lastly, an options system is in place such that visualizations can define options for their visualizations. These options can be things like how many layers of nodes to display, or a maximum number of nodes to show in the screen in general. These options can then be added onto the web-page at a later stage.

This web page will communicate with the visualization handler in order to show the actual visualizations, but it also has quite some code of its own in order to interact with it in a nice and intuitive manner. We make use of the jQuery library in order to connect the web-page elements to our code, as this is a bit more neat than working with the native system directly. The web page shows 4 quadrants that are all passed to the visualization handler such that it knows it can visualize data here. We also created our own jQuery plugin that allows us to make these areas resizable on the page. This system is then extended in order to add a button that hides 2 visualizations completely, such that the user can focus on 2 bigger visualizations if this is preferred by the user. Similarly we made use of this system to also allow the user to view a single visualization in full-screen mode, which will even get rid of the header of the page. For all these types of changes on the page

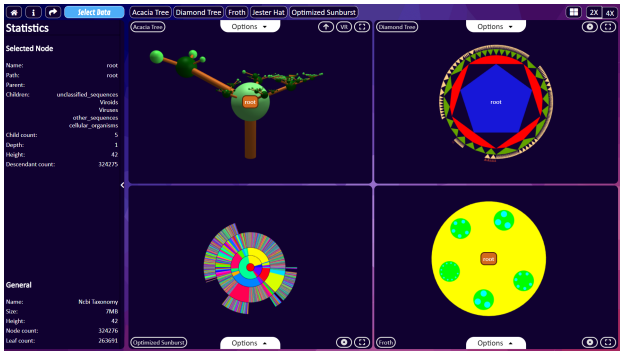
we have set up transitions using a combination of JavaScript and CSS code. These will nicely show what is happening instead of instantly changing the screen which could confuse people. The page also adds option panes to all 4 quadrants. These panes can be expanded in order to show the options that the visualization has defined and allow users to alter these. These options can also be buttons that perform tasks like focusing on the parent of a certain node, in which case you would not want to open the options pane constantly in order to use it. For that reason we have added the ability of defining an icon for your button. If an icon is added, it will instead appear as a quick button right next to the full-screen button. These buttons can be accessed way more quickly and are handy for common actions. Finally, the web-page has a sidebar that shows relevant statistic about the data set as a whole as well as the node that is currently focused or highlighted. This data is extracted from the synchronization data that the visualization handler stores.

5 Application Example

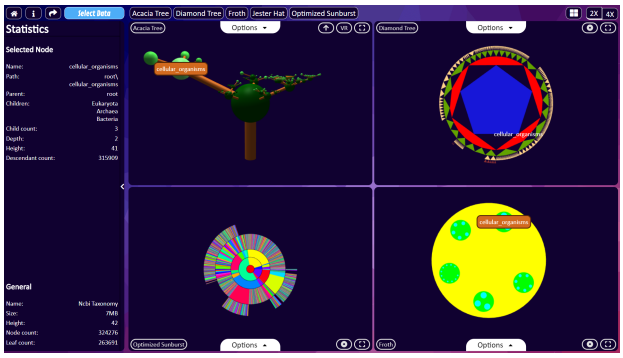
In this section we will illustrate and walk through a situation where the user loads the NCBI Taxonomy data set into VizWick.



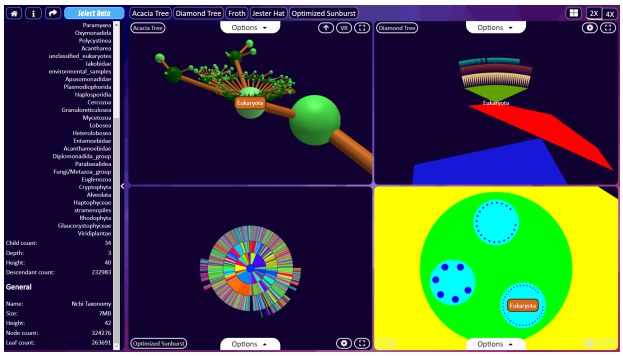
The NCBI Taxonomy is provided to users as an example data set on our web page. Suppose the user chooses to visualize this data set in the 4X view, using Acacia Tree, Diamond Tree, Froth, and Optimized Sunburst as methods of visualization.



The root node is selected by default when loading a data set. By looking at the statistics bar, the user immediately sees how large the NCBI Taxonomy is: the root has over 324 000 descendants in total, and 5 children. Moreover, the user notices a structure in this data set. From the Acacia Tree visualization, which shows a significantly bigger branch to the left than to the right, it is suggested that the hierarchy tree is quite unbalanced, with most of the 300 000 nodes lying in one of the root's 5 subtrees.



Intrigued by this finding, the user hovers over the biggest-looking node, called "cellular organisms", and discovers from the statistics bar that this node has a descendant count of over 315 000 and 3 children – that is almost as many as the root! It is clear now that the NCBI Taxonomy contains one very long path in its hierarchy tree. The user proceeds with selecting the "cellular organisms" node.



Doing so reveals yet another structure in the hierarchy tree. Out of "cellular organisms"'s 3 children, there is one called "Eukaryota" which has a stunning number of over 239 000 descendants and 34 children. This is also visible in the visualizations – for instance, Froth (bottom right) shows the 3 children of "cellular organisms", with "Eukaryota" containing the largest number of dark blue circles inside of it.

The illustrated situation is only one of the possible discoveries that the user can make using VizWick. It is notable that the statistics bar and the visualizations collaborate in showing the user the most out of the data set – in this case, initially when they noticed that there seems to be one very large subtree of the root, the statistics bar confirmed this fact by giving the large descendant count of that child.

6 Discussion and Limitations

There are several issues we are aware of. To start off, our version of Froth does not use circle packing, as this is a complex mathematical problem which we do not want to spend too many resources on. This means our version of Froth has a lot of empty space.
Diamond Tree also has a noticeable issue. When using a tree of a large amount of layers the area each node can occupy shrinks drastically, this means deep nodes appear almost as lines, making it unsuited to looking deep into trees.
Another issue that we discovered is that some visualizations differ greatly, making synchronizing of certain interactions, like focus or highlighting, hard or even impossible. These visualizations do not have a corresponding action to such a state, and thus the state is not reflected there.
There also exist several limitations with the framework as a whole, thus affecting all the visualizations. Each visualization is only capable of rendering a limited number of nodes at once, so only a segment of the data set can be shown at any one time. Instead, interactions can be used to navigate through the tree, still allowing one to observe all the information. for most data sets however, the entire hierarchy is

impossible to show all at once. It also means some data sets might not be rendered correctly, as when the amount of children of a single node exceeds the maximum nodes rendered, the excess will not be displayed. The amount of nodes the visualization can render depends mostly on the computer running the program, but around 500 is a realistic approximation. There are several reasons this approximation is this low:
Firstly, each visualization on screen is rendered at 30 frames per second, as this is the standard frame count for smooth movement. If we lowered this, the visualization would start looking jittery and feel slow to respond. However, even as many visualizations remain static most of the time, we cannot drop frames, our framework does not support this. this means we have to keep drawing the frames of a static visualization, even if nothing happens. Consequently, all visualizations on screen use up some amount of processing power at all times, which might detract some space from the visualization that currently needs it. With the virtual reality this aspect is magnified even further. Since VR requires 90 frames per second to run, the VR visualizations can usually only display around 200 nodes.
Secondly, we have not found proper resources for optimizing the PixiJS and Three.js rendering process. therefore we have only used the basic methods to render our visualizations. A better understanding of the libraries might result in some more options for optimization.
thirdly, some of the visualizations use simple physics to improve information conveyance or to improve aesthetics by animating them. Currently these physics are computed in JavaScript, making the process run through all the nodes sequentially. If we were to compute this for each node simultaneously however, we would be able to speed this up considerably. This would be done by computing animations via a graphics card using some OpenGL techniques. this would however require a lot of research, which would have cost more time than we had available for this problem.

7 Conclusion and Future Work

In about 9 weeks, we went from having nothing to developing a web-based tool that aims to visualize hierarchical data in the simplest and most interactive way possible. Our journey to VizWick had its ups and downs, however every member of our group learned something new and gained new experiences.

It all started by studying the existing literature of countless distinct visualizations and from them we selected eight. Our initial goal was to implement seven visualizations (one for each member), but because of the problems we had along the way we reduced it to the current number of visualizations that the user can find on VizWick, namely five. We wanted not to copy these visualizations, but to recreate them and improve them where possible and to add extra features in order to have a more intuitive result; and we did it – an example of that is the 'View the whole visualization' option which we added in the Diamond Tree and in Optimized Sunburst (originally Self-Adapting Sunburst): through this option you can view the whole data set regardless of what node is selected.

We decided to implement our own framework (instead of using a library like D3.js for example) because in this way we had more freedom in creating the visualizations and also on experimenting with them. Furthermore, this also helped us to gain a broader picture into the power of JavaScript, as well as to acquire a deeper understanding about the process of visualizing a hierarchical data set. Moreover, our visualizations are implemented as an interactive web-based tool, thus allowing the user to change different options and parameters about them.

VizWick is a great tool, but as everything, it is not perfect and can be further improved. If we are going to work with VizWick in the future, there are many things that could be added. The most important of which is to simply add more visualizations, and make sure that they vary a lot. And to support more variety of visualizations, we would also want to improve our framework. As mentioned in the limitations section, we currently render at a fixed frame rate. If we were to change this and make it dynamic, we could add support for visualizations that render a huge chunk of the data set in a second thread (or worker in web terms), and show the result afterwards. This would make for a less dynamic visualization, but it could give insights that our current visualizations can not. There are also a lot of other features that we would want to add however. Simply exporting the currently shown visualizations as an image would be very useful for instance. Of course the user can make a screen shot of his/her screen as it is, but this is a bit more limiting. If we were to make a proper export feature, we could render a larger part of the data set and allow the user to set the resolution of the image, and render in that resolution. This means that the user would not be limited by his/her computer performance and monitor resolution. Another very useful feature would be to have a dynamic list of all nodes of the data set, and make it queryable by different features like name, depth or size. This would greatly enhance the quick targeted exploration capabilities of VizWick. The final feature that we think is very important to add, is making the web page responsive. This means that the web page will correctly adapt and continue working on different smaller screen sizes, like phones or tablets. Currently the web site simply presents a message saying that it is incompatible with smaller screens, and presents the video to the user.

However, we as a group are very grateful that we came up with a tool like VizWick, which we hope is going to help people across the world to visualize hierarchical data sets.

References

[1] Li-Wei, Gong & Yi, Chen & Xin-Yue, Zhang & Yue-Hong, Sun. (2013). A Hierarchical Data Visualization Algorithm: Self-Adapting Sunburst Algorithm. 185-190. 10.1109/ICVRV.2013.3
[2] Lisong Sun, Steve Smith, Thomas Preston Caudell, "Interactive Poster: FROTH - a Force-directed Representation Of Tree Hierarchies", IEEE Symposium on Information Visualization, InfoVis 2003, 108-109
[3] Poster: Using Orthographic Projection and Animation to Convey Treemap Structure; https://blogs.sas.com/peerrevue/uploads/OrthographicTreeMap-summary.pdf
[4] IEEE Xplore; https://ieeexplore.ieee.org/document/6912178/
[5] Sender, B. and Sender, B. (2018). Newsweek Career Tree Visualization with RaphaelJS and Burst - data analysis, data visualization, design, tools and workflow - Bocoup. [online] Bocoup.com. Available at: https://bocoup.com/blog/newsweek-raphaeljs-career-tree-visualization-raphaeljs-burst [Accessed 27 May 2018]. [6] OneZoom, a fractal explorer for the tree of life http://journals.plos.org/plosbiology/article?id=10.1371/journal.pbio.1001406
[7] Generalized Pythagoras Trees for Visualizing Hierarchies (2014) http://www.scitepress.org/DigitalLibrary/Link.aspxdoi=10.5220/0004654500170028
[8] Wikipedia https://en.wikipedia.org/wiki/Hierarchical_database_model
[9] Wikipedia https://en.wikipedia.org/wiki/Newick_format
[10] Lou, Xinghua, Liu, Shixia, Wang, and Tianshu. (2008). FanLens: A Visual Toolkit for Dynamically Exploring the Distribution of Hierarchical Attributes. IEEE Pacific Visualisation Symposium 2008, PacificVis - Proceedings. 151 - 158. 10.1109/PACIFICVIS.2008.4475471.
[11] D3.js; https://d3js.org/
[12] Threejs.org. (2018). three.js - Javascript 3D library. [online] Available at: https://threejs.org/ [Accessed 27 May 2018].
[13] PixiJS; http://www.pixijs.com/
[14] Circle Packing; https://www.sciencedirect.com/science/article/pii/S0377221707004274