Chronogram Visualization for Tracking Honey Bees

1. Introduction

Honey bees have been a valuable topic in research because of their complex social behavior and their importance to our agriculture [8]. With our software, we aim to provide effective tools to analyze their behavior over a long extended period of time with video tracking and data visualization.

This software provides biologists easy to use tools that facilitate collecting and evaluating patterns of the behavior of bees. The interface itself has the ability to create and record annotations in its video interface by manually labeling bees. This helps save time and resources as collecting big data from thousands of bees can be time and mentally consuming.

The main tool for visualization is a chronogram graph that records and displays a frame log for when the bees enter and exit the colony. As the main tool to annotate the bees behavior, it helps the users visualize the bees' pattern better. This graphical visualization gives the user a clear overview of the bee behavior which is important to enable them to handle hours and hours of complex videos. The chronogram obtains marked common bee activities and displays them. As one of the research's main goal is to find new methods to analyze and understand bee behavior, this is an impertinent visualization for the researchers to use as evaluation.

We opted to update the web app with a friendlier user interface that has the ability for users to register and save analytical data in their accounts. We also wanted to add the ability for users to have access to multiple videos that record bees entering and exiting the honeybee colony. This meant using a front end framework that allow crowd source data analysis, and pairing it with a back end framework that will pull video data from a server, adding to the web interface client and server components.

Another feature that was added to this semester's research pipeline was automatic analysis for bee detection in the video interface, with the ability to differentiate the bee's activity (entering or exiting the colony, doesn't or does have pollen, is it fanning).

This report will focus mostly in explaining the methods that were used to update the functionality of the chronogram. This includes explaining the technologies that were used and how they were used, including API, libraries, functions and programming languages. It will describe the challenges that we had with the code and how we overcame them. It will also briefly explain the frameworks and API that were also used to turn the software into a client server system. At the end, the report will include the result of this semester's work, a conclusion and our expectations for future work.

2. Related Work

2.1 Past Work

2.1.1 Introduction to the Software Label Bee

The main goal for the Label Bee open source software is to develop a platform that analyses individual insect behavior and acquires new observations into the role of individual diverse behavior on the bee colony performance. It is meant for biologists to make annotations based on observations to joint videos, remote visualization and data acquisition that monitors thousands of marked bees over a extended and continuous period of time [6].

Since the web interface is meant to give access to an abounding amount of high definition videos, it is required we stored them in servers that have the capability to manage data from large datasets. That's why we will be using the resources from the High Performance Computing Facility at Puerto Rico. In the end, the software wants to provide semi-supervised machine learning that has the ability to manually and/or automatically detect different common activities known from honey bee such as pollen carrying or fanning behavior.

The web application that was developed last year already has many of the features mentioned previously. For the purpose of easier understanding, the software's component will be explained in the next section with visual representation of the interface.

2.1.2 First Label Bee Interface

The user's interface is meant to be simple and straightforward for users that either have software development experience or don't have programming experience. The

functionality of the its main elements are shown in Figure 1.

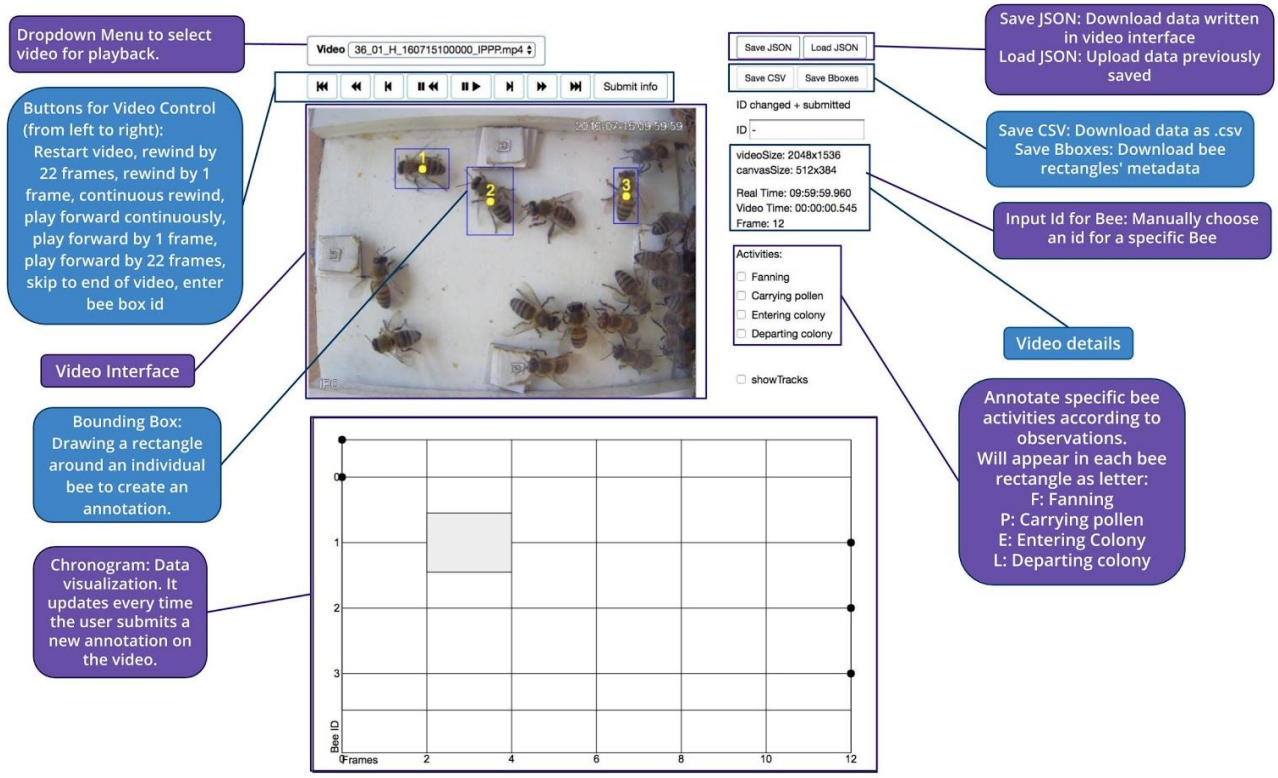

Figure 1: Parts of the software's interface with its description

2.1.3 Software Data Diagram

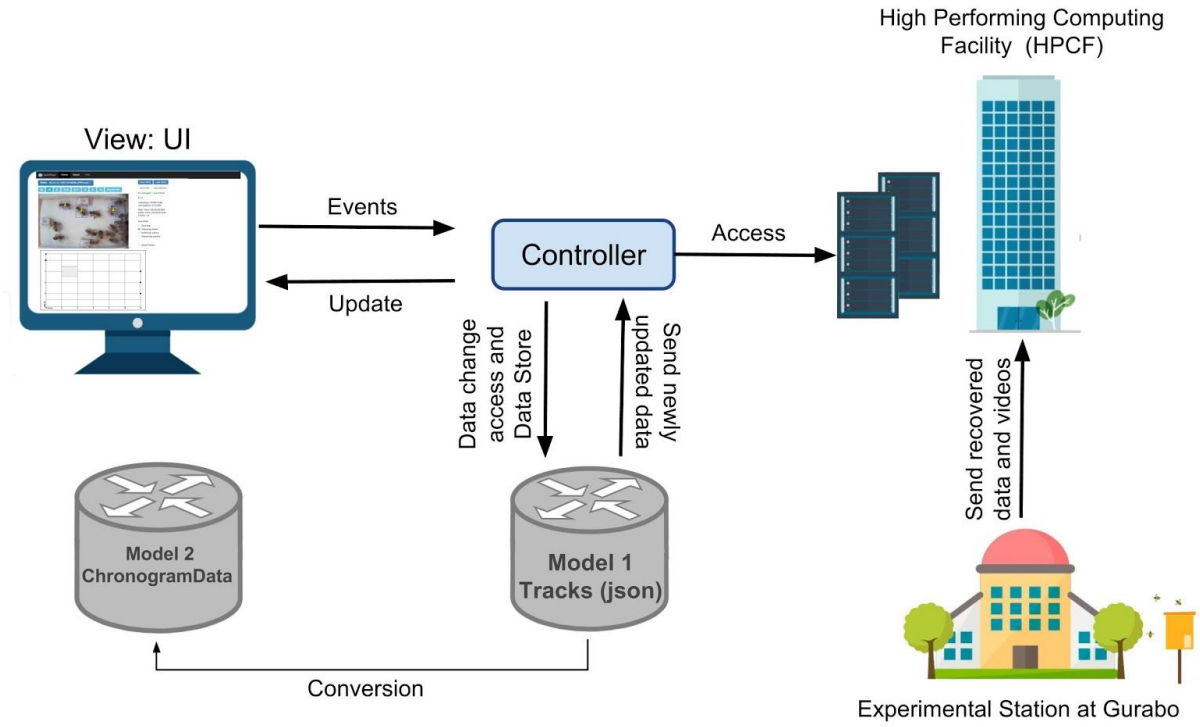

Figure 2: Label Bee Data Flow Architecture

The experimentation with the honey bee hive and data recovery it's first done at the Experimental Station at Gurabo. From there it is stored in our designated servers at the HPCF.

The data is managed through Model-View-Controller design pattern, as seen in Figure 2. The Model encapsulates the data and defines the logic and computation that manipulates the data. Because model objects represent knowledge and expertise related to a specific problem domain, they can be reused in similar problem domains. [3] The view is a visual representation of the model [1]. The view objects can acquire data from the model and allow the user to edit that data. A controller is the link between the a user and the system. It is through the controller that the View learns about the changes in the Model and vice versa.

In the case for the Label Bee software, the view is all of the software's user interface (UI). The input for the user is drawing a box around a bee on the embedded video. Once the info is submitted, that data is stored at the model to the Tracks data structure and then converted to chronogramData. The one that updates that data is the controller, which handles the input by binding the data in the form by different variables [8].

2.1.4 Label Bee Data Structure

A. Tracks

Tracks is the main model in the system because it makes all the functionality possible [8]. Once the user submits its first input, the controller creates the data structure with the annotations that were made. As the user keeps making more annotations on the video, the new data is being pushed to Tracks. This data is being saved in the browser until the user downloads it or the page reloads and it will be refresh into a new one. Tracks feeds the data to the other data structure Chronogram Data which will pass the data to be visualized in the chronogram.

```
[{"0":{"ID":"0","time":0,"frame":0,"x":840,"y":224,"cx":962,"cy":352,"width
":244,"height":256,"bool_acts":[false,false,false,false]}}]
```

Script 1: Example of Track data structure with one bee annotation

Script 1 shows how the data is saved in Tracks according to its description. To visualize it better, the data structure can be represented in a table.

Frame	Bee ID	Time	X coordinates	Y coordinates	cx coords	cy coords	width	height	Activities

Table 1: Representation of the Tracks Data Structure

All the rows highlighted in green are information of an individual bee. The x and y coordinate represents the x and y for the left up corner of the rectangle. The width and height are also from the rectangle that was made by the user. The element bool_acts represent the booleans for activities that was marked in an individual bee. This list of booleans will depend on wherever the user marks if the bee is doing a specific activity, as shown in Figure 1.

For now, Tracks is the only data structure that can be downloaded by the user directly from the interface.

B. ChronogramData

A chronogram is an inscription, sentence, or phrase in which certain letters express a date or epoch. The functionality for the chronogramData data structure is to have the data from Tracks organized and filtered by the needed elements to visualize the bees activities on the chronogram graph. When the Tracks data is created, so is the

chronogramData. In order to visualize every new change in the chronogram graph, chronogramData is always been restarted with the new input the user submits.

```
[chronoObservation =
   {Activity:"fanning"
    X:"0"
    Y:"0" }]
```

Script 2: Example of chronogramData Structure with on bee annotation

ChronogramData is a list of objects called chronoObservation. As seen on Script 2, the data that it recovers from Tracks is the bee's activity, the Bee ID which is the Y coordinate and the frame number which is the X coordinate.

2.2 Technology used

2.2.1 D3

The Data-Driven Documents or D3, is an open source JavaScript library for manipulating documents based on data and create data visualization. It is a fast and flexible API that supports large data sets and dynamic behaviors and interaction[2]. D3 also provides many built-in reusable functions and function factories, such as graphical primitives for area, line and pie charts. D3 works by allowing the user to bind data to a Document Object Model (DOM) and then apply data-driven transformations to create tables or interactive charts [8].

In this project, D3.js is used to build the chronogram display.

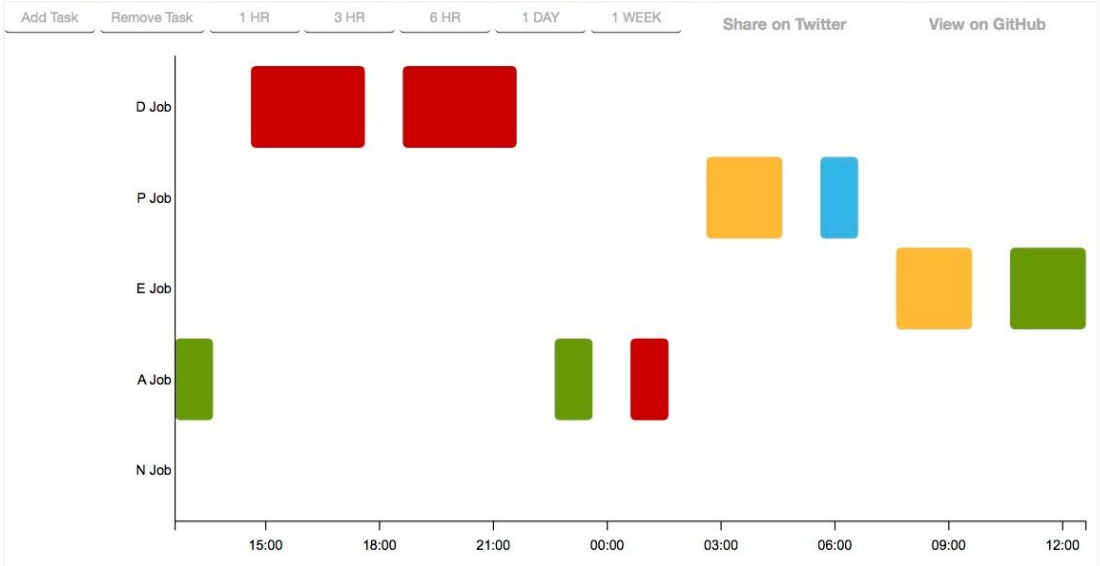

Figure 3: A Gantt Chart made with D3, which was taken as an example to build Label's Bee chronogram

2.2.2 Json

JSON or JavaScript Object Notation is a lightweight text-based open standard designed for human-readable data interchange[4]. Its is meant to exchange and store data between the browser and the server.

The data structure Tracks is downloaded as a JSON to display the annotations the user made. The user can also upload a JSON file into the interface with the same data structure as Tracks to display already saved data.

3. Methods

3.1 Main D3 objects used in the project

The D3 library allows us to manipulate elements of a web page in the context of a data set. These elements can be HTML, SVG, or Canvas elements. Depending on the type of visualization you want to create to represent your data, you can take advantage of D3 different functions and objects.

First, a SVG (Scalable Vector Graphic) canvas is created in the html page. This is the maximum dimensions of the height and the width that the chronogram graph is going to have. It is also the designated place on the html page that it's going to appear. Since the main goal for the chronogram data is to visually represent when the identified bees were in the video, the bee ID is going to be the Y coordinate and the frame number is going to be the X coordinate. The Y and X axis for the graph are created by adding each coordinate variable a scale function. Scales are a convenient abstraction for a fundamental task in visualization: mapping a dimension of abstract data to a visual representation [4]. In the case for the X axis (frames), since the data is continuous,a linear scale function is added to the variable. Linear scales are a good default choice for continuous quantitative data because they preserve proportional differences.

```
xScale = d3.scale.linear()
                .range([0, width])
                .domain([0, d3.max(chronogramData, function(d) {
                        return Number(d.x);})]);
```

Script 3: Code to add linear scale function to the X coordinate

Script 3 presents D3 programming format. After adding the linear scale function, the range and domain have to be specified. For the case of x, the range begins in 0 and ends in the width of the graph. For the domain function, it also begins on 0 but since it is dependant on the largest frame number, which is going to be constantly changing with the user's input, the maximum domain is going to have access to the data in

chronogramData. *D3.max* returns the maximum number in a given array, which in our case are the x values.

At the y axis, we only want to display the dee ID's that the user has specifically annotated, and not the numbers in between. In this case, since it's categorical data, the scale that will be added to the Y axis is ordinal scale, which specifies an explicit mapping from a set of data values to a corresponding set of visual attributes [d3 wiki].

```
var allIDs = []
yScale = d3.scale.ordinal()
            .domain(allIDs)
            .rangeRoundBands([0, height - margin.top - margin.bottom],
0.1);
```

Script 4: Code to create the Y Ordinal scale.

The domain function for the ordinal scale accepts an array of values. That's why the function is given a new array, which when the page is initialized it's empty. The function *.rangeRoundBands()* sets the scale's range to the specified array of values while also setting the scale's interpolator to interpolateRound, which returns an interpolator between the two numbers a and b and rounds the resulting value to the nearest integer [d3 wiki].

Once the scale variables have been made, it's pretty straightforward to create the axis in D3. The x and y axis variables are equal to the *d3.svg* elements, that also have the *d3.axis()* and *d3.scale()* function. The svg variable is then created and by writing svg = d3.select("#svgVisualize"), we specified to D3 where we want our svg elements to display in the html page. *D3.select* selects the first element that matches the specified selector string [d3 wiki]. In this case, we pass the string "#svgVisualize" as the parameter, which in the html script, it's an id. The width and height graph variables are added to the svg variable with the *d3.attr* function.

For organizational purposes, we create another svg element called chart, and append it to the already created svg element. This will help make the display of the graph more manageable. The y and x axis elements are appended to the chart element. The circle and rectangles shapes that represent the identified bees appearances are also appended to the chart element.

```
<svg id="svgVisualize" >
      <g class="display">
            <g class="x axis"> </g>
            <g class="y axis"> </g>
            <circle>
            <rectangle>
```

```
    </g>
</svg>
```

Script 5: HTML results of the svg elements created with D3 functions.

D3 functions converts the code into svg and css elements to display it in the View. The first element of "svgVisualize" is the canvas for the graph. The next element with the class display is the chart element that was appended to the svg canvas.

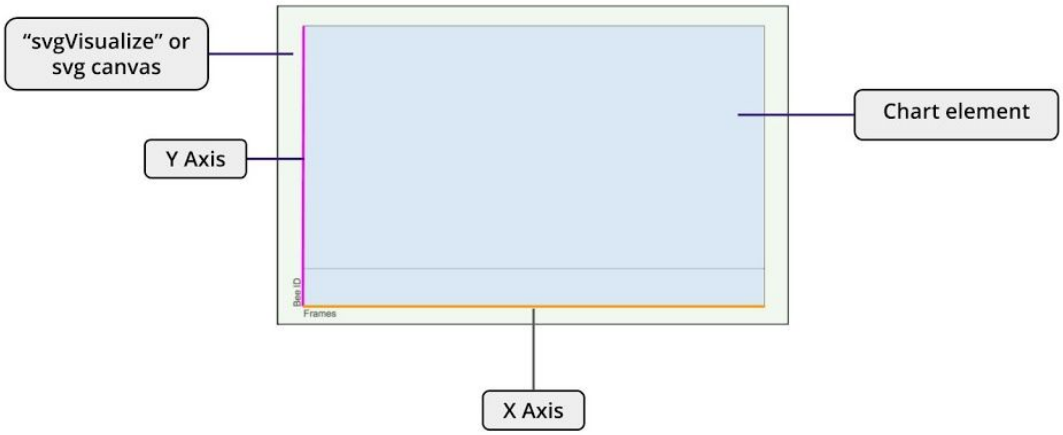

Figure 4: Svg elements for the chronogram

By appending the different svg elements, they become a sub-element of the svg canvas.

Before using *selection.append(string)* function to add the circles and rectangles to the chart element, we use *selection.selectAll(string)*, *selection.data(data)* and *selection.enter()*. At *Selection.selectAll(string),* for each selected element in *selection*, *selectAll* selects the descendant elements that match the specified *selector* string. The elements in the returned selection are grouped by their corresponding parent node in this selection [4]. For example, in the case of *chart.selectAll("circles")* selects all the elements with the attribute of circles in the svg chart. This is how it will be able to manipulate and change the circles based on the changing data. This data is specified on *selection.data(string),* which Joins the specified array of *data* with the selected elements, returning a new selection that represents the *update* selection: the elements successfully bound to data [4]. The *selection.enter() is* typically used to create “missing” elements corresponding to new data. The code used to create the circles and rectangles will be further explained in the following sections.

3.2 Sample based display

We based our first sample display of the chronogram to a scatterplot graph. A scatterplot is a useful summary of a set of bivariate data (two variables), usually drawn

before working out a linear correlation coefficient or fitting a regression line [7]. Each circle in the chronogram will visually represent in what frame each identified bee was.

After the graph axis and canvas have been initialized with its init functions, the chronogram has to be constantly aware of any changes that were made by the user. We create an update function that's going to be called every time there's new input data. This means that D3 is going to be re-drawing the contents in the chronogram with every change in the input. Since the x and y scale have to be updated with the newest maximum x or y value, the functions are given the new chronogramData x and y arrays. For the x scale, the line of code is going to be very similar to Script 3:

```
xScale.domain([0,d3.max(chronogramData, function(d) {return
Number(d.x);})]);
```

Script 6: X scale domain on the update function.

For the ordinal scale, it is not as simple. As mentioned before, we only wish to display the numbers who have been used to identify a bee, and omit the rest including duplicates. For this, before calling the domain function for the y scale and updating, we create a new set from the array of y values. Then, we convert this object to an array again and pass it to the y domain.

```
allIds = new Set();
for (let d of chronogramData) {
    allIds.add(d.y); }
yScale.domain([...allIds].sort());
```

Script 7: Y scale domain on the update function.

Since the circles are dependant of the constantly changing data from chronogramData, they are going to be initialized and updated in the global update function. For the first stage of testing the chronogram, we pass chronogramData to the *selection.data(data)* function to create the circles as polygons. After the circles are initialized, they have to be updated with the *cy, cx* and *r* attributes.

```
chart.selectAll("circle").data(chronogramData).enter().append("circle");
chart.selectAll("circle")
     .attr("cx", function(d) { return xScale(Number(d.x)); })
     .attr("cy", function(d) { return yScale(Number(d.y)); })
     .attr("r", 5);
```

Script 8: Functions to initialize and update circle

For Script 7, the string "cx" tells us to move x units from the left to the right of the SVG element. The "cy" string tells us to move y units from the top to the bottom of the SVG element. In this case, we are feeding the circles all of the x values in chronogram data, which are the frames, and for cy, we are passing all the y values which are the bee ids. The "r" simply stands for radius and it's the width of each circle.

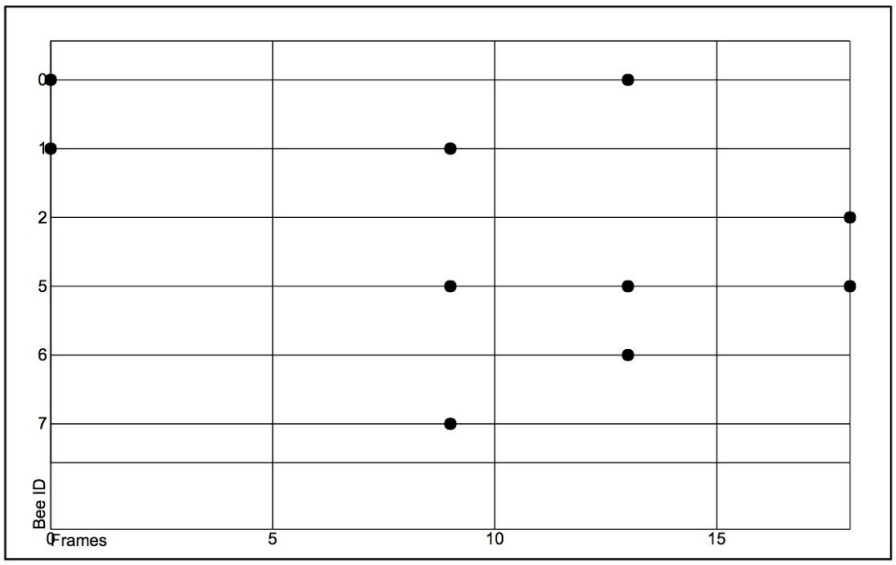

Figure 5: Example of sample display with circles

3.3 Conversion from samples to intervals

In order to add more variety to the chronogram visualization and help the users analyze the bee's activity diagram better, we decided to use different polygons to differentiate with the bees that had been identified in one frame versus the bees with the same id that had been annotated within a continuous set of frames. To keep the methodology simple, we opted to display an interval only when there was a sequence with a difference of exactly 1 between each frame number. This interval will be represented by a rectangle. If there was an annotated bee marked in multiple frames with the same id but it wasn't marked in a plus one pattern in the frames, it won't be considered an interval and it will be displayed as a circle.

To add a polygon or a shape to the scene with D3, you have to add all of its initial functions and its specific attributes to that polygon within the same block of code. So, in a way, it's very hard to add more than one polygon to the visualization within the same section of codes and functions. For us, this meant that we had to filter the data from the chronogramData data structure manually, and then pass is accordingly to each polygon. This made us create a new function, called createIntervalList to filter the coordinates from chronogramData and create the intervals.

A new temporary object was created that followed a hash design called tempCoordinates. In this object all of the different bee ids were pushed as a new object where the ids is its key and is values are a list of x values.

```
For i in chronogramData
    If i.y is in tempCoordinates already:
        Push current x values
    Else:
        Start new y object
        Push new object to tempCoordinates
```

Pseudocode 1: Algorithm for the creating of the temporary coordinate objects

The main goal for this pseudocode is to pair up all the xvalues together who have the same bee id.

The new data structure for intervals, which is a list of objects, is going to be created from tempCoordinates.

```
for (var key in tempCoordinates) {
    let xValues = tempCoordinates[key];
    let tempInterval = { x1: xValues[0], x2: xValues[0], y: key };
    for (var i = 1; i < xValues.length; i++) {
        if (xValues[i] - xValues[i - 1] == 1) {
            tempInterval.x2 = xValues[i];
        } else {
            allIntervals.push(tempInterval);
            tempInterval = { x1: xValues[i], x2: xValues[i], y: key };
        }
    }
    allIntervals.push(tempInterval);
}
```

Script 9: Code block that creates the new data structure of allIntervals

The tempInterval has three attributes, x1 which in the case of an interval is the minimum value, x2 value which is the maximum number in the interval and the y coordinate. The if condition compares the current x value with the next one to see if there is a plus 1 difference. If there is, the existing interval will be extended. If not, the current interval will be pushed to final data structure and a new tempInterval will be made.

AllIntervals is now going to become the data structure that is going to pass data to create the circles and the rectangles. In the case of the circles, apart from changing the data in the *selection.data(allIntervals),* there is going to be a condition at the

attributes functions for cx and cy where if x1 is the same as x2, then return x1. In the case for the rectangles, it is the opposite condition for its x and y attributes.

```
allIntervals = [ {x1:1,x2:1,y:0}, {x1:1,x2:3,y:1} ]
```

Script 10: allIntervals Example

4. Results

This updated version of the Label Bee's user interface provides the tools for the users to make multiple annotations based on observations on the recorded videos and have those inputs visualized graphically in an identification and frame relationship. The chronogram provides a visualization for solo bees and bees that had been annotated within an interval.

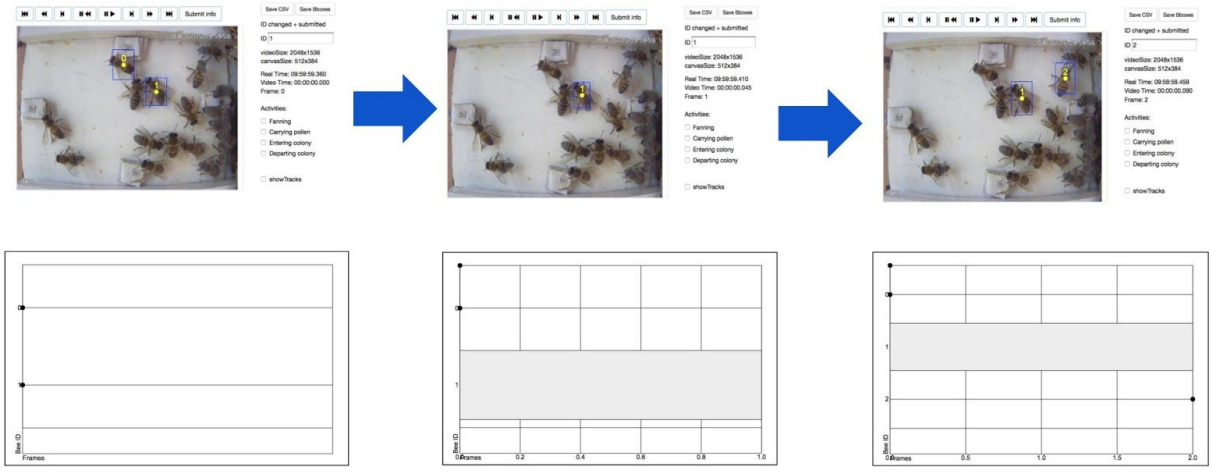

Figure 11: Example of user input and display of intervals in chronogram

Figure 6 shows an example of the user input on the video and the output in the chronogram. The user first marks two bees as 0 and 1 in frame 0. Obviously, none of these annotations are intervals, so they are represented as circles. In the next frame the user marks bee 1 again in frame 1. Since it's an interval, a rectangle is created with a x axis domain from 0 to 2. In frame 2 two bees are identified as 1 and 2. Since the bee with id 1 is still in interval, the rectangle expands to frame 3.

To properly display the intervals and coordinates as different polygons, another data structure was created. While this might make the conversion of data from Tracks to svg objects more long, it does make the data more manageable and organized. We can see the conversion process for the data in Figure 7.

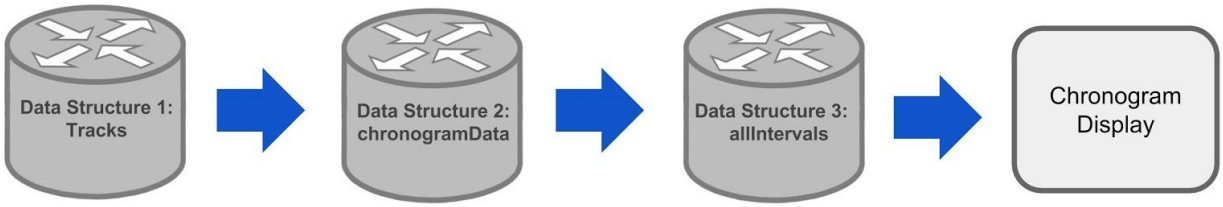

Figure 12: Data Structure Diagram

5. Conclusion

To add more functionality efficiently and keep the software tools simple, the Javascript code for the interface was studied, inspected and edited. The main feature added the in the code is the ability for it to filtered out data from chronogramData and create intervals for each identified bees. Instead of just having one polygon to represent all the identified bees, the chronogram displays two polygons to help the user more easily differentiate the activities of the bees in the video. For this, a new data structure was created that saves the x (frames) intervals.

Understanding another programmer's code with minimal comments while having a small amount of experience with Javascript and almost no experience with D3 library was a challenge by itself. The code had to be read multiple times and there were a lot of questions asked to our supervisor in order to fully understand what it was doing. The D3 documentation had to alse be read carefully as D3 has many functions of its own. One of D3 main powerful features is that it keeps track of what data belongs to what for you. However, it might not always have already made functions that filtered out more specific data, therefore the programmer would have to do it manually. This happen when we had to filtered out the x intervals from the chronogramData. Creating the algorithm to extract the x values and saved them as intervals in the new data structure was hard because it had to work for all different patterns the x values could had, including the situation where there was no interval.

In the process of updating the chronogram, more understanding towards the scripting language Javascript was obtained. More insight was gained towards D3's unique writing syntax and treating every attribute and function as an object. It was also learned the importance of analyzing a certain problem from afar, classifying an input and an output and decipher the best algorithm that will work in every case scenario. Jumping to guess different types of implementation without a clear understanding of the problem results in wasting more time and resources.

For the future, we wish to add more features to the chronogram, including having the ability to zoom to a precise part in the chronogram, delete already submitted data, be able to see the bee id when hovering on top of its polygons, and have visual representation of the common bee activities which are fanning, carrying pollen, leaving

and entering. While there was a visual representation of these activities already implementing in the code through different colors, it was concluded that this is not an efficient approach. When the user marks an activity, the polygon in the chronogram will appeared with the color that was assigned to the activity that was marked. However, if the user marks more than one activity, the polygon will be filled with the latest color on the queue. Therefore, the user won't have a visual representation of more than one activity. In order to improve this feature, we have to find a new way of visualization that will be easy to differentiate between activities and more than one activity can be represented in the same polygon.

6. Acknowledgements

We would like to thank Dr. Rémi Megret for helping us by providing the code, answering the many questions we always had and giving us the opportunity to work on this project. This work is supported by the National Science Foundation under Grant No.1707355 and 1633184.

7. Appendix

To further explore the updated code for the label bee interface, the link to the github repository is here:
https://github.com/gracemarod/chronogramLabelbee/blob/master/labelBee_v2.js.

8. Reference

[1] Atwood, Jeff. "Understanding Model-View-Controller." *Coding Horror*. N.p., 05 May 2008. Web. 05 June 2017. <https://blog.codinghorror.com/understanding-model-view-controller/>.
[2] Bostock, Mike. *Data-Driven Documents*. N.p., n.d. Web. 05 June 2017. <https://d3js.org/>.
[3] "Cocoa Core Competencies." *Model-View-Controller*. N.p., 21 Oct. 2015. Web. 05 June 2017. <https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html>.
[4] "D3 API Reference." *D3*. Github, n.d. Web. 05 June 2017. <https://github.com/d3/d3/blob/master/API.md>.
[5] "JSON Tutorial." *Www.tutorialspoint.com*. N.p., n.d. Web. 05 June 2017. <https://www.tutorialspoint.com/json/>.
[6] "Large-scale multi-parameter analysis of honeybee behavior in their natural habitat." *BigDBee Project*. N.p., n.d. Web. 05 June 2017. <http://bigdbee.hpcf.upr.edu/>.

[7] "Scatterplot." *Stat.yale*. Yale, 1997. Web. 05 June 2017. <http://www.stat.yale.edu/Courses/1997-98/101/scatter.htm>.
[8] Serralles, Shirley, and Remi Megret. "Bee Tracker." (2016): n. pag. Web. 5 June 2017. <https://drive.google.com/drive/u/0/folders/0B4JgJapKh__dbDlDRDFzQWlNcTQ>.
Chart from figure 2 is recovered from http://dk8996.github.io/Gantt-Chart/
Illustrations in figure 2 are recovered from https://www.vecteezy.com/, http://www.freepik.com and https://pixabay.com.
Illustrations in figure 12 are recovered from https://pixabay.com.