Design Document

SportsPA

Name:	Gerard Dobbs
Student Number:	C00196843
Course:	Bachelor of Science (Hons) Software Development
Supervisor:	Mr. Paul Barry
Date:	18/04/2018

Table of Contents

1. Introduction

This design document will detail the steps taken to achieve the goals specified for each iteration. The specifications were supplied by the Client, Mr. Johnny Bradley, and agreed upon with Paul Barry and myself. The specifications were supplied on an ongoing basis, making the project fluid in its nature. Due to the fluidity of the project, each of the three overall iterations were broken down into smaller ones. An XML file of events, supplied by the Client, was used during the course of the project. This file was of a hockey international match between Ireland and New Zealand.

The specification supplied, along with the tools, libraries and frameworks used to solve the problem for each iteration will be explained. The overall specification of iteration 1 was to produce a plot of each event that was recorded for each particular match. The iteration was broken down into 3 smaller iterations.

The first of these mini Iterations is simply to produce a count of each recorded event in the xml file.

The second mini iteration of Iteration 1 is to create a graph of the recorded events which would show the timeline of when they were recorded.

The final mini Iteration of Iteration 1 was to prepare for playing the associated video. In order to play the video JavaScript code would be needed. This code would need to be able to access the start and end times of each event. In preparation for this an alert box will be displayed with all the details of each event accessible upon clicking that event on the graph. This alert box will be removed in Iteration 2.

Iteration 2 will involve the playing of an associated segment of video when the user clicks on a recorded event depicted in the graph. Other functionality was added here to improve the user experience.

Iteration 3 involves developing a web application and deploying it to Glasnost. This is an added specification that will be provided to make the application accessible from anywhere and to provide storage for the files.

2.Project Plan

The project consists of three iterations of approximately six weeks in duration. The specifications and requirements for each iteration will be provided by the client Johnny Bradley. At the outset Johnny had a general idea of what the final product would look like. The overall, and very general, requirement can be seen in Figure 2.3.

This project is very fluid in its nature in that specifications would be supplied as each stage is completed. For that reason the first iteration was broken down into 3 smaller iterations.

After each iteration is completed, the result will be presented to the client for approval or otherwise. Any changes to be made or extra functionality required will be documented accordingly.

2.1 Iteration 1

Iteration 1 is broken down into three smaller iterations.

2.1.1 Iteration 1.1

The Specification supplied by Johnny Bradley, for iteration 1.1, was to be able to view a count of each recorded event. To achieve this the relevant XML file needed to be imported and parsed into a format suitable for generating the plot. The python library ElementTree would be used to achieve this and a graph of the counts produced.

The format of the XML files produced, when events are recorded, can be seen in Figure 2.1.1.1. In the supplied file, there were 623 similar instances displayed.

```
<instance>
<ID>2</ID>
<start>347.6330371010</start>
<end>360.3598275170</end>
<code>FRA A25</code>
<label>
<group>Quarter</group>
<text>Q1</text>
</label>
<label>
<group>Attack Entry</group>
<text>Right</text>
</label>
</instance>
```

Figure 2.1.1.1

The tag <ID> would not be needed throughout the project. The <start> and <end> tags are recordings of the start and end times of each event. These times are given in seconds. The <code> tag holds the name of the recorded events. These names would have been coded by students using GameBreaker, or similar software to record events. For this stage of the project, the clients specifications stated the the <label> tag and its child tags would not be used.

The recorded events are decided by the sports student. These may vary from match to match. For instance, in Figure 2.1.1.1, the student has decided that in this hockey match 'FRA A25' are events worth analysing. If it were a rugby match, it might be decided 'FRA Scrum' could be significant events.

A decision was made, when using ElementTree, to parse the XML file into many different python formats. Due to the fluid nature of this project, it was unclear what future requirements would include and consequently what formats would be needed. Therefore, by parsing the data into different lists, dictionary and sets, It was hoped that we would not have to return to this process during a subsequent iteration.

Having parsed the XML file into more usable formats, we could achieve the goal of this iteration which was to produce counts of each recorded events. To display the counts in a user friendly format the python Bokeh library would be used to create a graph. This library had been researched for the initial project idea as it was the most suitable library for producing real time interactive visualizations. It was decided to proceed with this library as we were unsure of future requirements and felt Bokeh would be able to meet any needs identified further into the project. Therefore it would be used to complete iteration 1.1, and display a graph of the event counts.

The python Bokeh library, which was chosen to display the graph, would need the data supplied to it to be in a pythonic format such as lists, sets or dicts. This was achieved by using ElementTree, which uses a hierarchical structure.

The first task would be to access the root of the tree. All instances, as shown in the example in Figure 2.1.1.1, are children of the <ALL_INSTANCES> tag. The root of the tree <file> and its immediate children, including <ALL_INSTANCES>, can be seen in Figure 2.1.1.2. Accessing <ALL_INSTANCES>, through the root of the tree, would then allow all other child tags to be accessed throughout the project.

```
<file>
    <SESSION_INFO>
        <start_time>2017-07-13 13:54:11.64 +0200</start_time>
    </SESSION_INFO>
    <ALL_INSTANCES>
```

Figure 2.1.1.2

The code to extract the <ALL_INSTANCES> tag, and its children is shown in Figure 2.1.1.3.

```
15      tree =etree.ElementTree(file = file)
16      root = tree.getroot()
17      rootInstance = root.find('ALL_INSTANCES')
```

Figure 2.1.1.3

Having accessed <ALL_INSTANCES>, it would then be possible to access any child tags. Due to the fluid nature of the project, it was decided to extract the data into different lists, sets and dicts. This was done so that as specifications were changed or added, any data required would be available in one of these formats.

The codes, start times and end times would be put into a list of dicts, and also into their own dictionaries, sets and lists.

```
113 for i in rootInstance.iterfind('instance'):
114     if i.findtext('code') not in codes_set:  # First time to see code
115         codes_set.add(i.findtext('code'))
116         temp_dict['code'] = i.findtext('code')  # Add new code to dictionary
117         codes_list.append(i.findtext('code'))
118         for j in rootInstance.iterfind('instance'):  # Loop through codes again
119             if i.findtext('code') == j.findtext('code'):  # Code == one in outer loop
120                 count += 1
121                 start.append(round(float(j.findtext('start'))))  # Add start time to list of starts for that code
122                 if float(j.findtext('start')) < plot_start:
123                     plot_start = float(j.findtext('start'))
124                 end.append(round(float(j.findtext('end'))))
125                 if float(j.findtext('end')) > plot_end:
126                     plot_end = float(j.findtext('end'))
127             each_code = [i.findtext('code')] * count  # Make list of the code x number of times it appears
128         all_codes.extend(each_code)  # Add the list of each code to list of all codes
129         temp_dict['count'] = count
130         codes_count.append(count)
131         temp_dict['start'] = start  # Add list of start times as value to start key in dict
132         temp_dict['end'] = end
133         code_list.append(temp_dict.copy())  # Add full dict to list
134         temp_dict.clear()  # Clear for next iteration
135         all_starts.extend(start)
136         all_ends.extend((end))
137         start = []
138         end = []
139         all_counts.extend([overall_code_count] * count)
140         count = 0
141         overall_code_count -= 1
```

Figure 2.1.1.4

All the codes, start times and end times were extracted first in the order that they were recorded and therefore as they appear in the XML file. They were then grouped together so that the lists could be used as data for the graphs to be produced.

The number of distinctive code types would also be needed for the graph. In the example used there were 38 separate codes. This would give the graph a height of 38 hbars on the y-axis. The first codes in the graph are ‘IRE Build Up’, as can be seen in ‘all codes’ in Figure 2.1.1.5. The ‘IRE Build Up’ would be displayed along the x-axis at a height of on the y-axis. The next row of codes would be would then be displayed at a height of 39. To achieve this the data for the graph plot would include a list of heights for each hbar. This is shown in the ‘all counts’ list of Figure 2.1.1.5.

```
start:  [538, 571, 610, 692, 717, 738, 798, 823, 884, 1151, 1201, 1246, 1355, 1406, 1497
end:  [561, 610, 624, 703, 732, 796, 804, 857, 894, 1173, 1241, 1282, 1370, 1427, 1523,
codes list:  ['IRE Build Up', 'Match Start/Stop', 'IRE Poss', 'NZL CP', 'IRE Turnover',
all codes:  ['IRE Build Up', 'IRE Build Up', 'IRE Build Up', 'IRE Build Up', 'IRE Build
all counts:  [38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38
```

Figure 2.1.1.5

In order to produce a graph of the number of each code, the list of all codes was iterated over and a new dictionary produced. The code to achieve is shown in Figure 2.1.1.6. A sample of the resulting dictionary is shown in Figure 2.1.1.7

```
22      for i in rootInstance.iterfind('instance'):  # Loop through each instance
23          if i.findtext('code') not in code_count:
24              code_count[i.findtext('code')] = 1
25          else:
26              code_count[i.findtext('code')] += 1
27      codes.append(code_count)
```

Figure 2.1.1.6

The code in Figure 2.1.1.6 loops through the XML file to find each code and puts it into a list of codes, if it has not already appeared in the xml file. Otherwise it loops through the file and counts the occurrences of that code. This produces a dictionary of each code and how many times each code appears, as shown in Figure 2.1.1.7.

```
code count {'IRE Build Up': 54, 'Match Start/Stop': 4, 'IRE Poss': 79, 'NZL CP': 20, 'IRE Turnover': 42, 'NZL GSO': 10, 'NZL A25': 54,
```

Figure 2.1.1.7

This satisfied the requirements specified at the start of this iteration. However, in order to produce the results in a more readable format, I produced a graph of the codes versus the count of each code. The codes for the x-axis, and the counts for the y axis of the graph were extracted from the dictionary, as shown in Figure 2.1.1.8.

```
33    list_keys = [k for k in code_count]
34    # or a list of the values
35    list_values = [v for v in code_count.values()]
```

Figure 2.1.1.8.

The code to plot the graph can be seen in Figure 2.1.1.9

```
39    p = figure(x_range=list_keys, plot_height=600, plot_width=1000,
40               title="Codes", toolbar_location=None, tools="")
41    p.vbar(x=list_keys, top=list_values, width=0.9)
42    p.xaxis.major_label_orientation = 1.2
43    p.xgrid.grid_line_color = None
44    show(p)
```

.

Figure 2.1.1.9.

The resulting graph is shown in Figure 2.1.1.10.

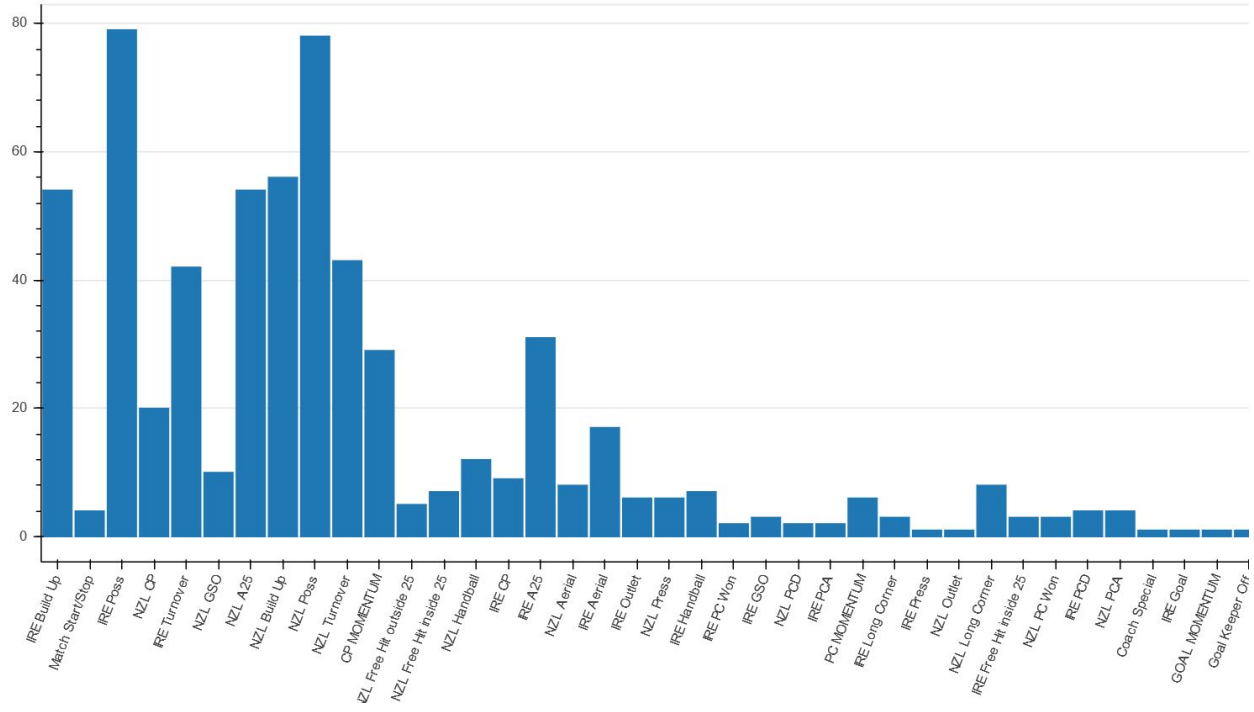

Figure 2.1.1.10.

2.1.2 Iteration 1.2

The specification of Iteration 1.2 entailed producing a graph which would include each recorded event. This would be similar to that of the SportsCode Elite software, which can be seen in Figure 2.1.2.1.

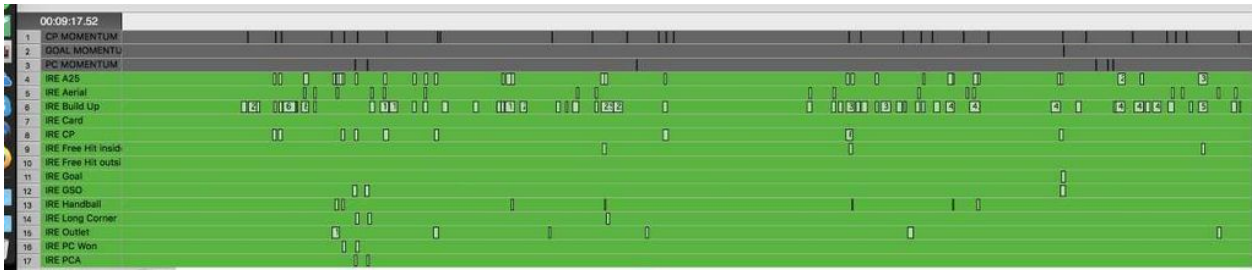

Figure 2.1.2.1.

Figure 2.1.2.1 displays the recorded events, including the start and end times. The codes can be seen on the y-axis. The x-axis is the timeline of the recorded match. Each bar in the graph represents a code, with the start of the bar being the start time of that recorded event and the end of the bar the end time. The next requirement, as part of iteration 1, was to reproduce a similar graph. The python Bokeh library was used in conjunction with the parsed XML file to meet this specification.

In order to achieve this requirement I used the Bokeh python library, which is described in the research document. Having upskilled myself in the use of Bokeh, I decided to use the rect glyph. This glyph is created by getting the centre point of the required rect, on both axes, and then using height and width to display the glyph. The centre point on the y-axis would correspond to the code, while the centre point on the x-axis would be the time halfway between the start and end time.

A function would be used to get the centre points as shown in figure 2.1.2.2, and a new list created.

```
#Get the mid point of each rectangle by getting average of start and end times
def centre(a,b):
    list = [(float(a[i]) + float(b[i]))/2 for i in range(len(a))]
    return list

#Get the width of each rect by subtracting end time from start time
```

Figure 2.1.2.2

The data frame ColumnDataSource is used to provide this data to Bokeh for plotting the graph. The necessary python lists, created previously would be supplied to this data frame as shown in Figure 2.1.2.3.

```
236     data = {'x': total_ends, 'y': temp_list1, 'left': total_starts, 'colors': colors, 'all_codes': total_codes,
237             'counts_list': counts_list, 'list_index_of_each_code': list_index_of_each_code
238     source = ColumnDataSource(data)
```

Figure 2.1.2.3

An example of the data supplied can be seen in Figure 2.1.2.4. The ‘left’ values supplied for the start of each hbar, on the x-axis, are the start times of each event. The ‘x’ values supplied to the end of each hbar are the end times. The ‘y’ values are represented by the number of distinctive coded events. In the example file used, the number of events is 38. This means all the first group of similar codes will be plotted first at 38 on the y-axis. The next group of codes will then be at 37 and so on. The final list supplied are the actual codes used for plotting.

```
left:  [538, 571, 610, 692, 717, 738, 798, 823, 884, 1151, 1201, 1246, 1355, 1406, 1497, 1646, 1759, 1779, 179
x:  [561, 610, 624, 703, 732, 796, 804, 857, 894, 1173, 1241, 1282, 1370, 1427, 1523, 1668, 1777, 1793, 1842,
y:  [38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 38, 3
all_codes:  ['IRE Build Up', 'IRE Build Up', 'IRE Build Up', 'IRE Build Up', 'IRE Build Up', 'IRE Build Up', '
```

Figure 2.1.2.4

It was known at this stage that the lists produced in the previous iteration would have to be updated in order to create the graph. In plotting the graph rects would be used to represent each event, which would be represented on each row of the graph. The graph would be built from the top down. The top, and first row, would show all the events of the first event stored in the XML file. This means that if ‘IRE Poss’ was the first event in the XML file then all ‘IRE Poss’ events would be shown in this row. Similarly the next event in the XML file, which has not already been graphed, would be shown in the 2nd row down, and so forth.

This layout for the graph means that the lists will have to be rearranged. All the ‘IRE Poss’ events previously stored in a list as they were recorded would have to be moved to the beginning of the list and similarly with the next event.

This would also have to be done for the ‘start’ and ‘end’ times. All the ‘start’ times for ‘IRE Poss’ would be moved to the start of that list. These times would be represented on the x-axis. The ‘start’ time would show the start of each hbar and the ‘end’ time the end of that bar.

The plot which would contain the graph of the hbars is then created. This was achieved as shown in Figure 2.1.2.5.

```
xdr = DataRange1d()
ydr = DataRange1d()
plot_height = 550
plot = Plot(
    title=None, x_range=xdr, y_range=ydr, plot_width=1100
    , plot_height=plot_height - 200,
    h_symmetry=False, v_symmetry=False, min_border=0,
    toolbar_location="right", background_fill_color="green")
```

Figure 2.1.2.5

The next step is to create the rects and add them to the plot. This is achieved by creating a columnDataSource which contains all the data needed to create the graph. The ‘data’ variable contains all the necessary lists. A columnDataSource data frame is the created of this data and assigned to the variable ‘source. This can be seen in Figure 2.1.2.6.

```
data = {'x': total_ends, 'y': temp_list1, 'left': total_starts, 'colors': colors, 'all_codes': total_codes,
        'counts_list': counts_list, 'list_index_of_each_code': list_index_of_each_code,
        'list_of_all_labels': total_labels, 'converted_start_times': converted_start_times,
        'converted_end_times': converted_end_times, 'converted_time_length': converted_time_length}
source = ColumnDataSource(data)
```

Figure 2.1.2.6

The rect glyphs are then created and inserted onto the plot using the data frame which was assigned to the source variable.

As part of this iteration a further requirement was to colour code the events. As the sample XML file used was a hockey match between Ireland and New Zealand, the colours used were Lime and black. Lime was used as the graph background would be green. Any event not associated with either team would be coloured white.

In preparation for this second specification, to colour code the teams, a list of corresponding colours would need to be generated. The python function, shown in Figure 2.1.2.8 creates the data to achieve this.

```
92      def assign_team_colors(t1, t2):
93          for code in all_codes:
94              if t2 in code:
95                  colors.append("Black")
96                  team_list.append(team2)
97              elif t1 in code:
98                  colors.append("Lime")
99                  team_list.append(team1)
100             else:
101                 colors.append(("White"))
102                 team_list.append('Neither')
```

Figure 2.1.2.8

This produces a list of colours, ordered as described earlier for the codes and times.The list produced to achieve the color coding can be seen in Figure 2.1.2.9.

```
['Lime', 'Lime', 'Lime', 'Lime', 'Lime', 'Lime', 'Lime', 'Lime', 'Lime', 'Lime', 'Lime', 'Lime', 'Lime', 'Lime', 'Lime', 'Lime', 'Lime'
```

Figure 2.1.2.9.

As can be seen from Figure 2.1.2.9, the colours are grouped together in the list. This is because all list elements are grouped as they appear in the graph. Therefore when a new element appears this represents a new row in the graph. The ‘Lime’ grouping shown above represents the first row of the graph which was an event relating to Ireland.

The elements in the ‘code’, ‘start’ and ‘end’ lists were grouped similarly. The result was the specified graph using hbar’s to display each event. The graph displayed each event and the start and end times of when each event occurred. A section of the resulting graph is shown in Figure 2.1.2.10, with the full graph shown in Figure 2.1.2.11.

Figure 2.1.2.10.

Having consulted with Johnny it was decided that the teams should be grouped together also. This again involved rearranging the lists used in the ColumnDataSource. The result is shown in Figure 2.1.2.11.

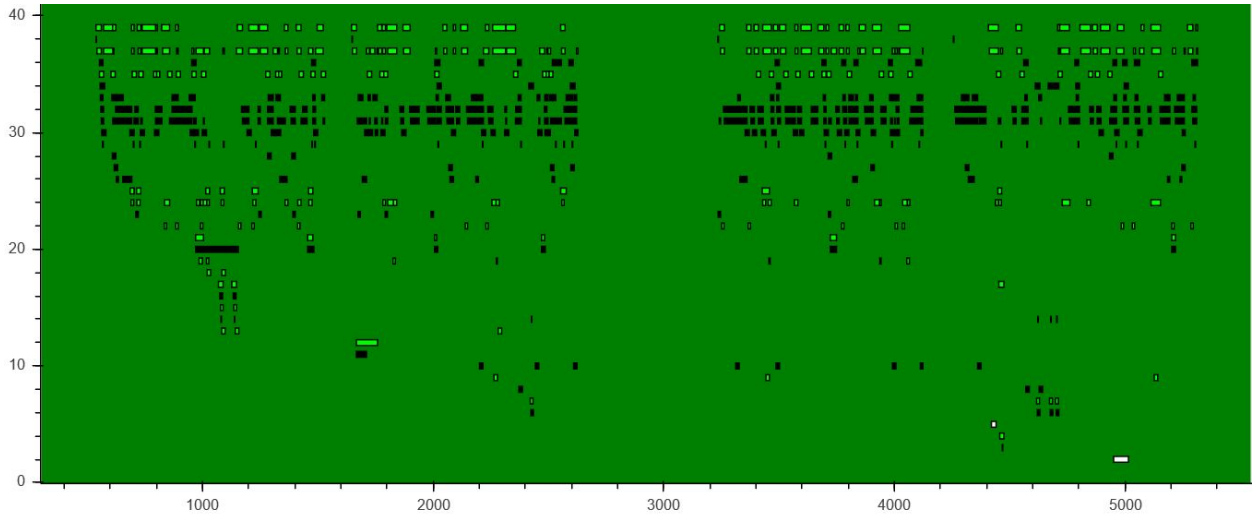

Figure 2.1.2.7

2.1.3 Iteration 1.3

The next requirement agreed upon was to make each bar clickable.The requirements specified the ability to be able to click on a hbar to play the relevant video segment. This ability to be able to click a hbar and produce an alert box was used as preparation for the next iteration. The alert box displayed the start and end times of that particular event. These times would be needed to start and end the video at the required video segment. The alert box would be removed in iteration 2

On commencing this requirement I discovered that using the rect glyph would not be the most suitable, for making the hbars clickable. I then had to change the code to use the hbar glyph in order to successfully complete this iteration. In order to make each bar clickable I used the Bokeh Tap Tool. This tool required implementing a 'CustomJS' callback function. This function was coded in JavaScript to display an alert box which would show details of the event, the start and end times, and the team associated with the event. The event code, start time and end time will be used in iteration 2 to play the associated video clip. During this iteration I also added a Bokeh Hover Tooltip, which was created using HTML.

The first task carried out was to include a Bokeh Taptool. The Taptool would be used to interact with the hbars. Once a hbar is clicked, the goal, in this iteration, is to display the start and end times in an alert box. The TapTool was first added to the plot, as shown in Figure 2.1.3.1.

```
taptool = TapTool(callback=callback)
plot.add_tools(taptool)
```

Figure 2.1.3.1

The callback function named callback is used to display the alert Box. This function can be seen in Figure 2.1.3.2.

```
callback = CustomJS(args=dict(source=source), code="""
      var data = source.data,
       //gets id of selected point ie. index of point created
       selected = source.selected['1d']['indices'],
       select_inds = [selected[0]];
   if(selected.length == 1){
       // only consider case where one glyph is selected by user
       selected_x = data['x'][selected[0]]
       selected_y = data['y'][selected[0]]
       selected_left = data['left'][selected[0]]
       selected_color = data['colors'][selected[0]]
       selected_code = data['all_codes'][selected[0]]
       selected_team = data['team_list'][selected[0]]
       var team
       if (selected_color=='Lime'){
           team='Ireland'
       }
       else if (selected_color=='Black'){
           team="New Zealand"
       }
       else{
           team="Neither"
       }
       window.alert("Start = "+selected_left+"ms , End = "+selected_x+"ms , Code = " +
        selected_code+" , Team = "+team)
   }
   """)
```

Figure 2.1.3.2

The callback function uses CustomJs in order to allow javaScript to be used to display the alert box. The data used to generate the plot, which is stored in a ColumnDataSource object source, is passed into the callback function as a parameter. This allows the javaScript access the data. The other parameter is the javaScript code used.
In the Javascript code in Figure 2.1.3.2, the relevant data relating to the clicked hbar is assigned to the variable ‘selected’. The individual selected data elements are then each assigned to their own variables. An example of this is ‘selected_left’, which will hold the left side of the hbar. This

'selected_left' is the start time of the video. These values are then displayed in the alert box, as shown in Figure 2.1.3.3.

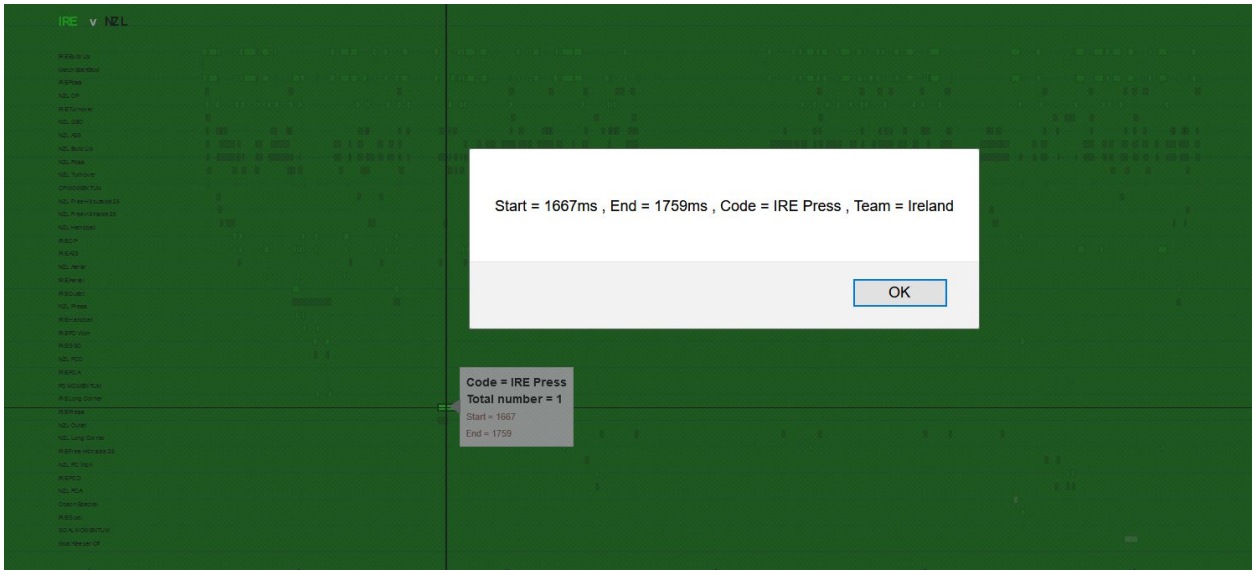

Figure 2.1.3.3

Extra specifications provided in this iteration were tools to be used in the graph. These tools included a pan tool, select tools and zoom tools. A hover tool producing tooltips was also included. This hover tool displayed details of the event, including the coded event name, the start and end times, the total count of that event in the file and the location of that event. This extra specification would prove very beneficial for graphs with a lot of events, where it would be difficult to read graph details.

Other Bokeh tools were then added, as can be seen in Figure 2.1.3.4. Thes tools were added above the graph on the right hand side as seen in Figure 2.1.3.5.

```
plot.add_tools(ResetTool())
plot.add_tools(hover)
plot.add_tools(BoxZoomTool())
plot.add_tools(CrosshairTool())
plot.add_tools(ZoomOutTool())
plot.add_tools(ZoomInTool())
```

Figure 2.1.3.4

Figure 2.1.3.5

The tools must be activated by clicking on them first. In Figure 2.1.3.5, the first two tools, BoxZoomTool() and TapTool have been activated.

The ResetTool() is used to reset the graph after other tools have been used on it. For example, when the TapTool is used it dims out the graph except for the clicked hbar. The ResetTool() will then reset the graph.

The other tool that was coded using javaScript was the hoverTool, which would display a tooltip. This tool would be very beneficial in viewing the data for each event before clicking. The javaScript code for the hoverTool can be seen in Figure 2.1.3.6. To access the data '@' is used before the name of the data element.

```
hover = HoverTool(tooltips="""
        <div bgcolor="#E6E6FA">
            <span style="font-size: 10px; font-weight: bold;">Code = @all_codes</span></br>
            <span style="font-size: 10px; font-weight: bold;">Total number = @counts_list</span></br>
            <span style="font-size: 8px; color: #966;">Start = @left</span></br>
            <span style="font-size: 8px; color: #966;">End = @x</span></br>
        </div>
    """)
```

Figure 2.1.3.6

Figure 2.1.3.7 and Figure 2.1.3.8 show the tooltip displayed when the cursor hovers over an event.

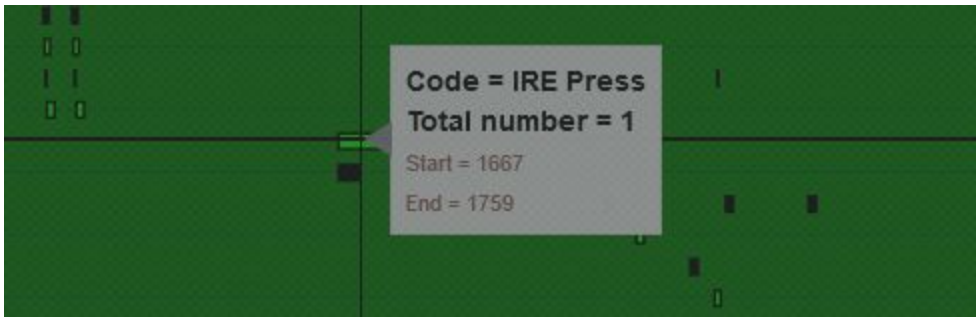

Figure 2.1.3.7

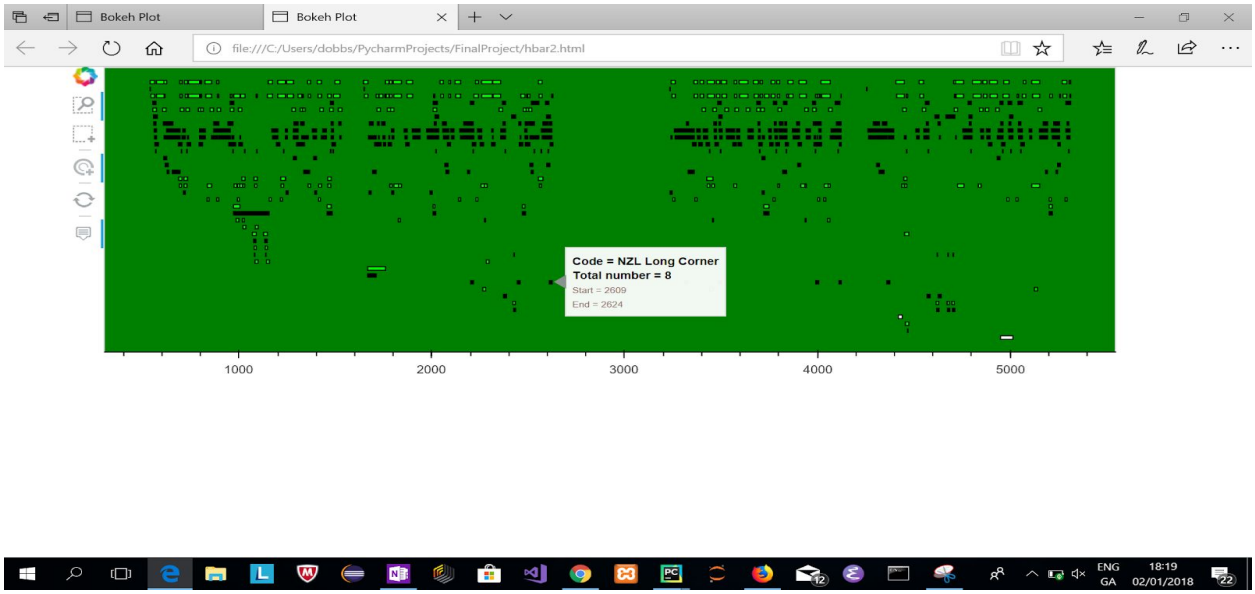

Figure 2.1.3.8

This completed all requirements supplied at the beginning, and during, iteration 1.3. The results were displayed to Johnny Bradley, who was extremely delighted with the results, particularly the extra requirements which were not available in previous software.

2.2 Iteration 2

Iteration 2 involves the ability to use the graphs hbars to play the associated segment of the relevant video. Research was carried out into different options available to satisfy this need. I looked at openCv and MoviePy in python and also the html5 video tag. Along with the research carried out I also spent some time experimenting with the 3 options. In the end i decided to go with the html5 video tag. The main reason for this was because I could use a CustomJS callback function to select the start and end points in order to play the relevant video section. I had upskilled and gained some knowledge in using this function while displaying the alert box in the previous iteration.

In Iteration 1.2 I had used an alert box to display the data that would be needed in this iteration. Having been able to access this data I no longer needed the alert box.

The first task was to create a div to contain the video tag. This can be seen in Figure 2.2.1.

```
479     """Create a html Div to contain the video"""
480     div = Div(text=f"""<video id="myVideo" width="540" height="310" controls>
481                 <source id="mySrc" src= {Video_file} type="video/mp4">
482                 </video>""", width=540, height=310)
483
```

Figure 2.2.1

The most challenging part in this iteration was the ability to pass the python video file into the html tag. This proved an even bigger challenge than I originally anticipated. That is until I discovered Literal String Interpolation, also known as f-strings. F-strings are available since python 3.6 and they provide a way to embed expressions inside string literals[1]. As can be seen from Figure 2.2.1, f-strings allowed me to embed the python video file into the html5 video tag.

Having include the video tag, the next goal was the ability to play sections of it, when a hbar was clicked. As stated earlier this would be by using the Bokeh tap tool. This tap tool uses a CustomJS callback function and the video needs to be passed into that function as seen in Figure 2.2.2.

```
523     taptool = TapTool(callback=display_event(div, divStart, divEnd, divEvent, divEventNum, divLabel))
```

Figure 2.2.2

The callback function calls a CustomJS function called ‘display_event()’. The first parameter passed to this function is ‘div’. We can see from Figure 2.2.1 that div includes the video tag. On the click of a hbar this function is then called. The signature of this function can be seen in Figure 2.2.3.

```
336     def display_event(div, divStart, divEnd, divEvent, divEventNum, divLabel, attributes=[]):
337         style = 'float:left;clear:left;font_size=32.5pt'
338         return CustomJS(args=dict(div=div, divStart=divStart, divEnd=divEnd,
339                                   divEvent=divEvent, divEventNum=divEventNum, divLabel=divLabel,
340                                   source=source), code="""
```

Figure 2.2.3

The display event function will retrieve the relevant data associated with the click. The first thing the function does, as seen in Figure 2.2.4, is to assign the columnDataSource passed to variable ‘data’. This data includes all the lists used in plotting the graph

```
var data = source.data,
```

Figure 2.2.4

When a click event occurs the corresponding elements in the data lists will be retrieved. This is achieved using the code in Figure 2.2.5.

```
347             selected_x = data['x'][selected[0]]
348             selected_y = data['y'][selected[0]]
349             selected_left = data['left'][selected[0]]
350             selected_color = data['colors'][selected[0]]
351             selected_code = data['all_codes'][selected[0]]
352             selected_list_index_of_each_code = data['list_index_of_each_code'][selected[0]]
353             selected_counts_list = data['counts_list'][selected[0]]
354             selected_converted_start_times = data['converted_start_times'][selected[0]]
355             selected_converted_end_times = data['converted_end_times'][selected[0]]
356             selected_converted_time_length = data['converted_time_length'][selected[0]]
357             if(data['list_of_all_labels'][selected[0]]=="")
358                 selected_list_of_all_labels = "NO LABELS";
359             else
360                 selected_list_of_all_labels = data['list_of_all_labels'][selected[0]];
```

Figure 2.2.5

If we look at line 349 in Figure 2.2.4, as an example, we see that the function assigns the selected element in the in the ‘left’ list to variable selected_left. The ‘left’ list contains the left of each hbar and subsequently the start time of that event. Therefore ‘selected_left’ represents the start time of the selected event to be played. Similarly the other details of the event are retrieved.

The start and end times need to be converted to seconds in order to play the video from the right place. This is done using the code in Figure 2.2.6.

```
362             var date = new Date(null);
363             date.setSeconds(selected_left);
364             start = date.toISOString().substr(11, 8);
365             var date = new Date(null);
366             date.setSeconds(selected_x);
367             end = date.toISOString().substr(11, 8);
```

Figure 2.2.6

Variables ‘start’ and ‘end’ now represent the video segment we want to play, according to the button click. The video is played using the code segment in Figure 2.2.7.

```
394             var vid = document.getElementById("myVideo");
395             vid.currentTime = start;
396             vid.play();
397             vid.addEventListener("timeupdate", function() {
398                 if (vid.currentTime >= end) {
399                     vid.pause();
400                     start = 0;
401                     end = 90000;
402                 }
```

Figure 2.2.7

In the code in Figure 2.2.7, after the video has been started an eventListener is used to check if the ‘end’ time has been reached, in order to stop playing.

This fulfilled the requirements outlined by Johnny at the start of this iteration, and again he was very pleased with the outcome. It was then decided that some data from the XML file, not used up to this point should be displayed. These tags were called ‘Labels’ and contained child tags, but not every event contained ‘Labels’.

Again I had to extract this data and put it into lists to include in the columnDataSource. The data would be displayed in the resulting html page beside the video. It was also decided to display all relevant data here also. Div’s were created to display the data and the empty div’s were passed as arguments into the ‘display-events’ function as seen in Figure 2.2.3. The code in Figure 2.2.8 is used to display the data.

```
divStart.text = "Start = "+selectedStart+"   ";
divEnd.text = "End = "+selectedEnd+"   ";
divEvent.text = "Event = "+selectedCode;
divEventNum.text = "Event No. "+selectedEventNum+" of "+timeList.options.length;
divLabel.text = "Label: "+selected_list_of_all_labels;
```

Figure 2.2.8

It was proposed to Johnny that in order to provide better usability to the end user that another option to play the video could be added. Looking at it from a non user viewpoint, it was proposed to add other means to play the video. This would include dropdowns to select and play an event, as well as a replay button. The idea behind adding these was to provide different options and an improved user experience. The graph contains a lot of hbars, as in the case of the sample XML provided which had 628 events and therefore 628 hbars. Finding certain events in the graph could prove difficult. Allied to this is the fact that some events are quiet short and therefore can be hard to click. The added functionality gives a much friendlier and faster means to find some events.This proposal was greeted with enthusiasm by Johnny.

The first dropdown box would include a list of all the exclusive coded events. The user would select an event from that dropdown and then the second dropdown would populate with the times available for that event only. Again this meant creating more lists of data.

This proved quiet time consuming as I ran into some difficulties along the way.

After doing some research I decided to use the Bokeh server to update the dropdown. I created the dropdowns as shown in Figure 2.2.9.

```
486     """Create the dropdown Select boxes"""
487     Times = Select(title="Times", id="time", value=startAndEndTimes[0][0], options=startAndEndTimes[0],
488                    callback=callbackTimes)
489     Codes = Select(title="Codes", id="code", value=exclusive_codes_list[0], options=exclusive_codes_list,
490                    callback=callbackCodes)
```

Figure 2.2.9

This was done firstly in a separate file and python code was used to update the 'Times' dropdown. After successfully getting it to run I introduced the code into my main file. It was at this stage that I realised, that using the Bokeh Server was causing issues when playing the video. Solving this was proving very problematic and time consuming already. It was decided to abandon this approach before any more time was lost.

The second approach taken was to update the dropdown on the client side using the DOM. This was proving fruitful, however in my haste to make up lost time, I failed to notice there were issues. The 'Times' dropdown was updating successfully on selection of an event in the 'Codes' dropdown. However, on selection of a time in the 'Times' dropdown the times in this dropdown were reverting to the original default values. It was as if the browser was refreshing on selection in the dropdown. Again I spent time trying to solve this issue, before deciding to take a different approach.

The third approach taken was to again use CustomJs. I had considered this approach originally but did not use it as I foresaw an issue with passing the data from the dropdown into the function. I sought assistance online but to no avail. As has been the case throughout this project the Bokeh docs support was not up to date. Similar problems had been raised but the solutions were outdated. I persevered in finding a solution and eventually found a way to achieve what i needed. One short line of code, as shown in Figure 2.2.10 was my saviour.

```
492     """Send Times dropdown to callback for codes"""
493     callbackCodes.args["s2"] = Times
```

Figure 2.2.10

The CustomJs function to update the 'Times' dropdown is shown in Figure 2.2.11.

```
    """Callback to update details in Time Select when Event is Selected"""
    dataSelect = {'y': startAndEndTimes}
    sourceSelect = ColumnDataSource(dataSelect)
    callbackCodes = CustomJS(args=dict(source=sourceSelect, divEnd=divEnd, divEvent=divEvent), code="""
        var data = source.data;
        var codeList = document.getElementById("code");
        var starts = data['y'][codeList.selectedIndex];
        s2.set('options', starts);
        s2.trigger('change');
    """)
```

Figure 2.2.10

The CustomJs function to play the video on selection of a time from the ‘Times, dropdown is similar to that shown in Figure 2.2.5 and subsequently explained.

Another suggestion made was to provide a replay button. This button would allow the user to replay the last event or to return to the start of the current event while it is being played. This would provide a better experience for the user as they would not have to search for the event again if further analysis was needed or if something was missed. The code for the callback function used for this button is shown in Figure 2.2.11.

```
"""Function for the Replay Button"""
callbackButton = CustomJS(args=dict(source=sourceSelectTime,div=divButton,divStart=divStart,divEnd=divEnd), code="""
    var divStartTime = divStart.text.substring(8,16);
    var divEndTime = divEnd.text.substring(6,14);
    var startSplit = divStartTime.split(':');
    var endSplit = divEndTime.split(':');
    var start = (+startSplit[0]) * 60 * 60 + (+startSplit[1]) * 60 + (+startSplit[2]);
    var end = (+endSplit[0]) * 60 * 60 + (+endSplit[1]) * 60 + (+endSplit[2]);
    var vid = document.getElementById("myVideo");
    vid.currentTime = start;
    vid.play();
    vid.addEventListener("timeupdate", function() {
        if (vid.currentTime >= end) {
            vid.pause();
            start = 0;
            end = 90000;
        }
    }, false);
""")
```

Figure 2.2.11

In this function, the data to be used is retrieved from the displayed data relating to the event already started or played.

Another functionality added in this iteration was to colour code the background. This would be used instead of colour coding the hbars. Different shades of the background would be used for codes belonging to each team and also neutral codes. The first task was to identify the dividing points along the y-axis for the colour change. This was done by using the code in Figure 2.2.12.

```
230     """Get the details of where the colour bands change"""
231     pos_count = 0
232     for code in total_codes:
233         if team1 in code:
234             pos_count += 1
235     top_lower = int(temp_list1[pos_count])
236     pos_count = 0
237     for code in total_codes:
238         if team2 in code or team1 in code:
239             pos_count += 1
240     bottom_higher = int(temp_list1[pos_count])
```

Figure 2.2.12

The variables ‘top_lower’ and ‘bottom_higher’ identify the points on the y-axis where the background colour changes. These variables are then used in Bokeh Box Annotations, which are used to create bands on Bokeh plots, as shown in Figure 2.2.13.

```
517 low_box = BoxAnnotation(top=bottom_higher, fill_alpha=0.3, fill_color='Green')
518 mid_box = BoxAnnotation(bottom=bottom_higher, top=top_lower, fill_alpha=0.5, fill_color='#00CC00')
519 high_box = BoxAnnotation(bottom=top_lower, fill_alpha=0.4, fill_color='Lime')
520
```

Figure 2.2.13

A title identifying the names of the teams involved was also added. The team names were taken from the title of the XML file as shown in Figure 2.2.14.

```
11  file = 'IREvNZL Events.xml'
12  team1 = file[0:3]
13  team2 = file[4:7]
14  title = file[0:7]
```

Figure 2.2.14

This code extracts the title and team names. The team names were extracted in order to use a different colour for each team in the title. Bokeh Text was used to display the title, as shown in Figure 2.2.15, and was added to the plot as shown in Figure 2.2.16.

```
551     home_title = Text(x=-470, y=count_used_for_title_height + 2,
552                       text=[team1], text_font_size="8pt",
553                       text_font_style="bold",
554                       text_alpha=1.0, text_font='helvetica',
555                       text_color=team_color[0], text_baseline="alphabetic")
556     versus = Text(x=-330, y=count_used_for_title_height + 2,
557                   text=['v'], text_font_size="8pt", text_font_style="bold",
558                   text_alpha=1.0, text_font='helvetica',
559                   text_color='White', text_baseline="alphabetic")
560     away_title = Text(x=-280, y=count_used_for_title_height + 2,
561                       text=[team2], text_font_size="8pt", text_font_style="bold",
562                       text_alpha=1.0, text_font='helvetica',
563                       text_color=team_color[1], text_baseline="alphabetic")
```

Figure 2.2.15

```
564     plot.add_glyph(home_title)
565     plot.add_glyph(versus)
566     plot.add_glyph(away_title)
```

Figure 2.2.16

The names of each event were also added to the plot along the y-axis. Bokeh Text was again used to achieve this. A list of the codes was iterated over and a Bokeh Text created starting from the top of the plot. An offset, determining where the Text is positioned, was increased as each Text was positioned. The code to achieve this is seen in Figure 2.2.17.

```
542     yaxis_offset = 0.5
543     code_index = len(codes_list) - 1
544     for code in exclusive_codes_list:
545         text = Text(x=-470, y=yaxis_offset,
546                     text=[exclusive_codes_list[code_index][0:30]], text_font_size="5pt",
547                     text_font_style="normal",
548                     text_color="White", text_baseline="alphabetic")
549         plot.add_glyph(text)
550         yaxis_offset += 1.01
551         code_index -= 1
```

Figure 2.2.17

The resulting graph can be seen in Figure 2.2.18, and the overall output is shown in Figure 2.2.19.

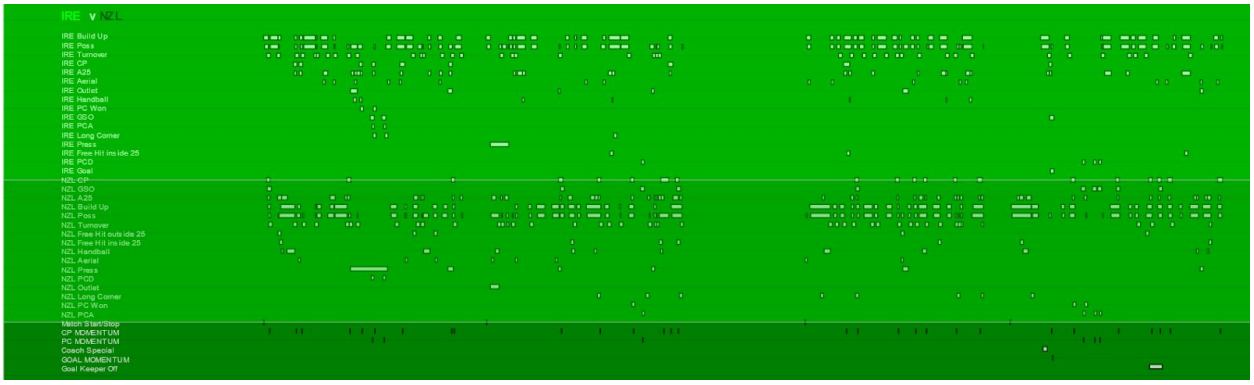

Figure 2.2.18

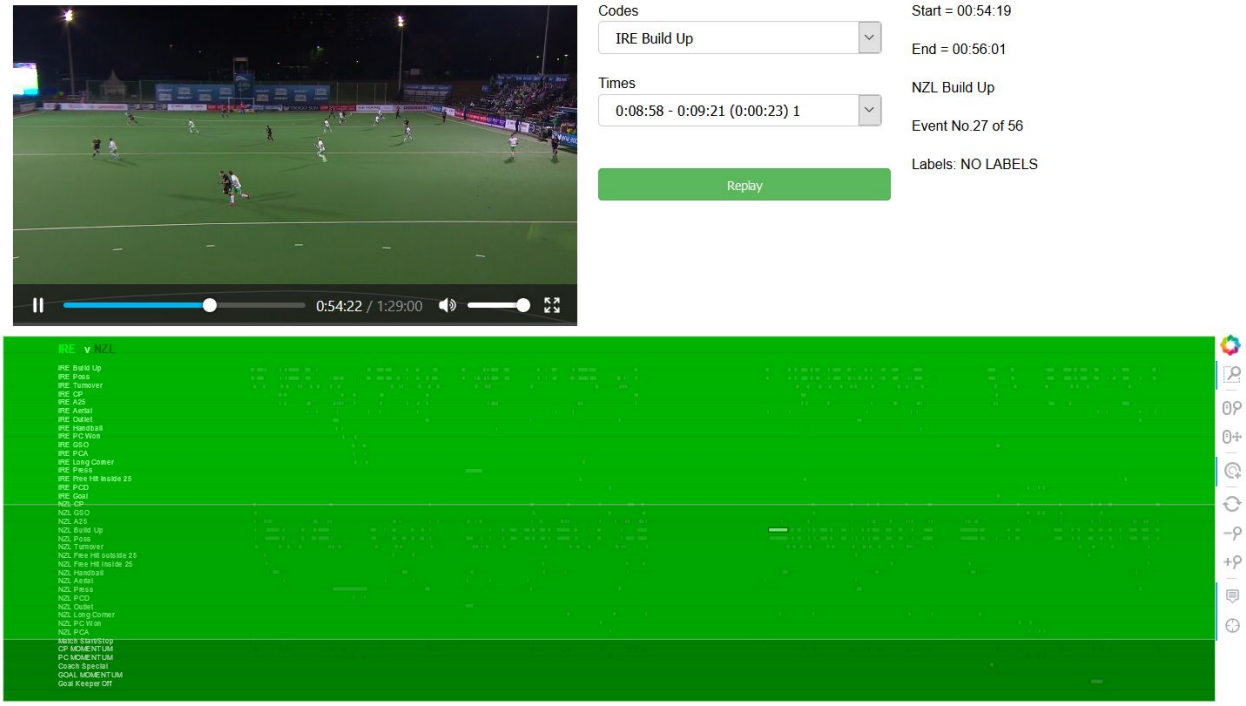

Figure 2.2.19

This completed iteration 2. Again the result was shown to Johnny Bradley, with a very positive response.

2.3 Iteration 3

Having completed the core functionality, Iteration 3 involved creating a web application. This web application would allow an administrator, such as Johnny Bradley to upload videos and Xml files. A user would then be able to select files to analyse.

It was decided to use Flask to develop the web application. I had thought about using Django. The reason I used Flask was because I had used it in my third year project and wanted to get even more experience in its use.

The first page to be created is the home page. This is where students would select the mp4 and xml files to do their analysis. There would be no need for students to log into an account to do this. They would be presented with 2 dropdowns from which they could select the files. A tab on this page would bring an administrator to a login page. Bootstrap was used for styling the web application.

The dropdown boxes would be populated with the files previously uploaded by an administrator. When the user navigates to the home page it calls the function shown in Figure 2.3.1.

```
609     """Read XML & Video files from static folder"""
610     video_path = 'static/video'
611     video_list = []
612     xml_path = 'static/xml'
613     xml_list = []
614     video_listing = os.listdir(video_path)
615     xml_listing = os.listdir(xml_path)
616     for infile in video_listing:
617         video_list.append({'name': infile})
618     for infile in xml_listing:
619         xml_list.append({'name': infile})
620     return render_template('login1.html', error=error, video_data=video_list, xml_data=xml_list)
```

Figure 2.3.1

This function gets all the mp4 and xml files stored in their respective folders and puts them into 2 lists. The html page is then rendered and the 2 lists passed to it as arguments.

The html page then creates 2 dropdown boxes and a play button. The dropdown boxes are populated from the lists passed in as arguments. An example of how this is achieved for one dropdown is shown in Figure 2.3.2.

```
<div class="row">
    <div class="col-md-6">
        <div class="form-group">
            <label for="xml_select">Select an xml file:</label>
            <select name="xml_select" id="xml_select" class="selectpicker form-control">
                {% for o in xml_data %}
                <option value="{{ o.name }}">{{ o.name }}</option>
                {% endfor %}
            </select>
            <div class="help-block with-errors"></div>
        </div>
    </div>
    <div class="col-md-12">
        <input type="submit" class="btn btn-success btn-send" value="PLAY">
    </div>
</div>
```

Figure 2.3.2.

As can be seen, the list of xml files is looped through in order to populate the dropdown box. When the selections have been made and the play button clicked, the form goes to the 'play' function, as seen in Figure 2.3.3.

```
<form method="POST" action="{{ url_for('play') }}">
```

Figure 2.3.3.

It is this function which then carries out the parsing of the data, the creation of the plot and allows the user to select what event, and subsequently what segment of video to play.

The home page also allows an administrator to go to a login page. Logging in allows the administrator to upload mp4 and xml videos. A successful login will bring the user to the upload page. The route for the upload page can be seen in Figure 2.3.4.

```
@app.route("/upload", methods=['GET', 'POST'])
@login_required
def upload():
    """Read XML & Video files from static folder"""
    video_path = 'static/video'
    video_list = []
    xml_path = 'static/xml'
    xml_list = []
    video_listing = os.listdir(video_path)
    xml_listing = os.listdir(xml_path)
    for infile in video_listing:
        video_list.append({'name': infile})
    for infile in xml_listing:
        xml_list.append({'name': infile})
    """Get path to project"""
    app_root = os.path.dirname(os.path.abspath(__file__))
    """Get a list of the files from form to be uploaded"""
    for file in request.files.getlist("file"):
        """Get file name from file object"""
        filename = file.filename
        name, ext = os.path.splitext(filename)
        if ext == '.mp4':
            target = os.path.join(app_root, 'static/video/')
        else:
            target = os.path.join(app_root, 'static/xml/')
            destination2 = "/".join([app_root, filename])
            file.save(destination2)
        destination = "/".join([target, filename])
        file.save(destination)
    return render_template('upload.html', video_data=video_list, xml_data=xml_list)
```

Figure 2.3.4

The files that are already loaded are first displayed on the page. This is done to inform the user if they are trying to upload a previously uploaded file. The function then gets a list of the files selected for upload. It checks the file extensions in order to decide which folder they are stored in and are then stored in their respective folders.

The administrator will also have the option of deleting files after login. This is achieved by selecting the delete tab. The administrator is presented with a dropdown list of all the stored files which can be deleted. Figure 2.3.5 shows the code for displaying the available files for deletion.

```
730     def delete1():
731         """Read XML & Video files from static folder"""
732         error = ""
733         video_path = 'static/video'
734         video_list = []
735         xml_path = 'static/xml'
736         xml_list = []
737         video_listing = os.listdir(video_path)
738         xml_listing = os.listdir(xml_path)
739         for infile in video_listing:
740             video_list.append({'name': infile})
741         for infile in xml_listing:
742             xml_list.append({'name': infile})
```

Figure 2.3.5

The delete() function searches the xml and video folders for files stored in them and puts them into to lists. The lists are then passed as parameters in a render_template function as seen in Figure 2.3.6.

```
return render_template('delete.html', error=error, video_data=video_list, xml_data=xml_list)
```

Figure 2.3.6

This will then display the html page with a dropdown containing all the files available for download. The html code to achieve this can be seen in Figure 2.3.7. The code loops through both lists to populate a drop down box with the files.

```
<div class="form-group">
    <label for="file_select">Select a Video file:</label>
    <select name="file_select" id="file_select" class="selectpicker form-control">
        {% for o in video_data %}
        <option value="{{ o.name }}">{{ o.name }}</option>
        {% endfor %}
        {% for o in xml_data %}
        <option value="{{ o.name }}">{{ o.name }}</option>
        {% endfor %}
    </select>
    <div class="help-block with-errors"></div>
</div>
```

Figure 2.3.7

The administrator then selects a file for deletion and clicks delete. The selected file is passed to the delete() function to variable ‘file_select’, as can be seen in Figure 2.3.8.

```
try:
    file_select = str(request.form.get('file_select'))
    name, ext = os.path.splitext(file_select)

    if ext == '.mp4':
        delete_path = video_path
    else:
        delete_path = xml_path
    if os.path.isfile(delete_path+"/"+file_select):
        os.remove(delete_path+"/"+file_select)
```

Figure 2.3.8

The extension of the file is checked in order to get the correct path to the file to be deleted. This sets the ‘delete_path’. The selected file is then removed and deleted. If an error occurs the administrator is informed, as can be seen in Figure 2.3.9.

```
except:
    error = "Deletion Unsuccessful"
    return render_template('delete.html', error=error, video_data=video_list, xml_data=xml_list)
```

Figure 2.3.9

3. References

Reference:
[1] https://www.python.org/dev/peps/pep-0498/