Shijie Zhao

Data Visualization for Mobile Game Design and Feature Selection

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor’s Thesis

08/05/2017

Author(s) Title Number of Pages Date	Shijie Zhao Data Visualization for Mobile Game Design and Feature Selection 50 pages + 5 appendices 8 May 2017
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Sakari Lukkarinen, Senior Lecturer, Metropolia UAS Joel Julkunen, CFO, GameRefinery Oy

The project explored new ways of visualization of mobile game analysis data. The aim was to find better approaches of visualizing analysis results to help the clients in mobile game design and feature selection decision making. This study was carried out for GameRefinery Oy.

The project included researching data visualization, comparing different visualization tools, implementing game analysis visualizations and conducting a usability test of the visualizations. Technologies used in the project include Python, Pandas and D3.js.

As a result, two visualizations of mobile game genre trends and features that were very different from other visualizations within the company SaaS (software as a service) and the mobile game industry were developed and were proven to be easy to use and helpful.

The data visualizations provided a clear and easy way to acquire related data and helped making fact-based decisions in game design process for the clients in the mobile game industry, where time is of the essence.

Keywords	Data Visualization, JavaScript, D3.js, Python, Pandas, JSON, Mobile Game Design, Feature Set

Contents

Abbreviations and Terms

Bokeh	A Python interactive data visualization library
CAD	Computer-aided design
Chart.js	A JavaScript light-weight data visualization library
CSS	Cascading Style Sheets
CSV	Comma-separated values
DataFrame	A 2-dimentional labeled data structure from Pandas library
DOM	Document Object Model
D3.js	Data-Driven Documents, a JavaScript data visualization library
GIF	Graphics Interchange Format
GPS	Game Power Score
HTML	Hypertext Markup Language
I/O	Input/output
JavaScript	A high-level, dynamic, untyped, and interpreted run-time language
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
MATLAB	A multi-paradigm numerical computing environment
Matplotlib	A plotting library for Python and NumPy library
NumPy	A Python scientific computing library
Pandas	A Python library of high performance data structures and analysis tools
PDF	Portable Document Format

Plotly	A scientific graphing library
PNG	Portable Network Graphics
Python	A high-level general-purpose programming language
SaaS	Software as a service
SUS	System Usability Scale
SVG	Scalable Vector Graphics
TeX	A typesetting system
TSV	Tab-separated values
UA	User Acquisition
XML	Extensible Markup Language

1 Introduction

This project explored new ways of visualization of game analysis data. The goal of the project was to find better approaches of visualizing analysis results to help the clients in decision making regarding mobile game design and feature selection.

The mobile game industry is growing rapidly on a daily basis, and the global games market will reach $108.9 billion in 2017 with mobile taking 42% [1]. Markets in China, US and Japan are quite different from each other and the competition is fierce with very rapid changes of game and genre trends.

The case company in this study is GameRefinery Oy, a mobile game analytics company that provides market specific insight into feature and implementation aspects of mobile game design, and helps the clients increase their games' profitability potential by implementing fact-based game design. The company is developing a SaaS (software as a service) platform, and is considering different ways of presenting the analysis data to the clients. The most valuable aspect of the company is the analysis data, and it is very important to use data visualization to present the data to the clients in a way that is easy and clear to perceive.

The objective of the project is to develop new ways of visualizing game analysis data that helps the clients find out current genre trend and most important game features quickly and conveniently, and make fact-based design decisions with their findings from the visualizations. The project managed to deliver a solution meeting the requirements and is proven to be effective.

The thesis starts by studying data visualization, comparing different data visualization tools and selecting one tool suitable for the project. Then exploring different visualization approaches for presenting mobile game analysis data that can be perceived fast and effortlessly. Matters such as the functionalities of the visualization tools, the environment of the visualizations and the usability of visualizations need to be taken into consideration.

2 Background

2.1 Data Visualization History

Quantitative information and its graphical representation has its origin deeply inside the history of thematic cartography, statistical graphics and data visualization, three fields that are associated with each other [2].

The earliest use of graphical representation appeared in geometric diagrams and in maps for helping navigating and exploring. Visualization saw its beginning of development by 16th century when techniques and instruments for precise observation and measurement of physical quantities were well-developed. In the 17th century, the emergences of analytic geometry, theories of errors of measurement, the birth of probability theory, demographic statistics and “political arithmetics”, marked the starting point of practice and great development in theory. During the 18th and 19th centuries, social, moral, medical and economic statistics - numbers related to people - were collected in a large scale and periodic manner. Moreover, the value of these data for planning and governmental response, or even as a subject by itself to be studied, started to be realized. [2]. Figure 1 is a historical astronomical chart created by Chinese cartographer Huang Shang in 1247 [3].

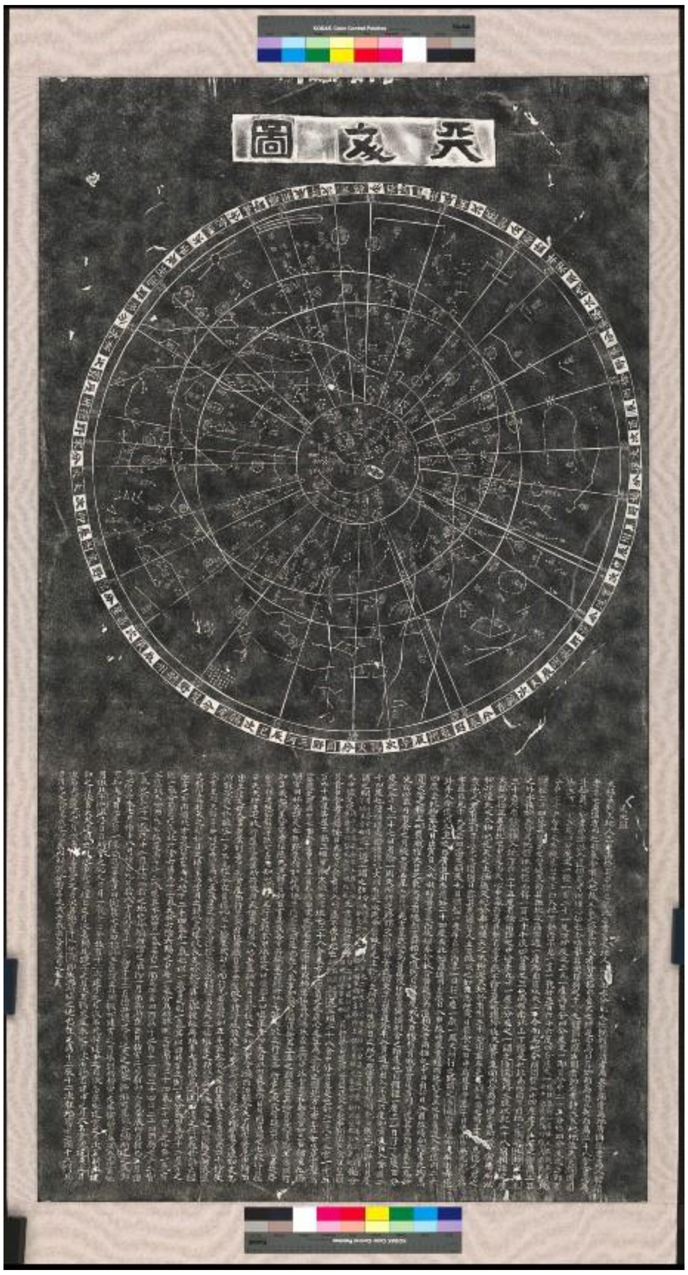


Figure 1.Tianwen Tu (Astronomical chart)

As can be seen in figure 1, it is a planisphere along with text explanations below it. With the help of a adujstable disk to cover some parts of the chart on top, it displays visible stars for any time and date, assisting in recognizing stars and constellations [4].

Statistical thinking started with the rise in visual representation: mathematical proofs and functions were illustrated using diagrams; nomograms were invented to help calculations; for easier communication and visually accessible representation of properties of empirical numbers - their trends, tendencies and distributions, several graphic forms were developed. Besides, the close relation of numbers of the state (the origin of the word “statistics”) and its geography aided it to be visually represented on maps, which is called “thematic cartography” nowadays. [2]

It has been always hard to generate maps, diagrams and graphs, and even harder to publish them. Originally they were hand drawn, later engraved on copper plate and manually colored, after that came lithography and photo-etching. Till most recently, computer programs have been used to generate graphs, even though the process is still constrained by the available technology. [2]

The rapid development of statistical computation and graphic display in recent years have provided tools for data visualization that was impossible half a century ago. The advancement in human-computer interaction has also contributed to brand new pattern for interactive graphical information exploration. Besides most of the progress in statistical data visual representation, information visualization itself has also evolved, especially for large networks, hierarchies, data base representation and so forth. [2]

2.2 Varieties of Data Visualization

As data visualization is constantly developing, and is being used in a wider range of fields, some branches of data visualization are becoming increasingly important and are worth researching by themselves. In this chapter, information visualization, scientific visualization, mathematical visualization and domain specific visualization are presented.

2.2.1 Information Visualization

Information visualization applies to the visualization of large-scale collections of non-numerical, non-coordinate, more abstract data, and it relies heavily on processing of abstract data into a more concrete form that can be perceived by an observer more effectively. Some examples are file systems and codes in software, library and bibliographic database and relation networks on the internet. [2] Figure 2 is a visualization of GDP data of cities from Zhejiang, Guangxi and Jiangsu provinces of China.

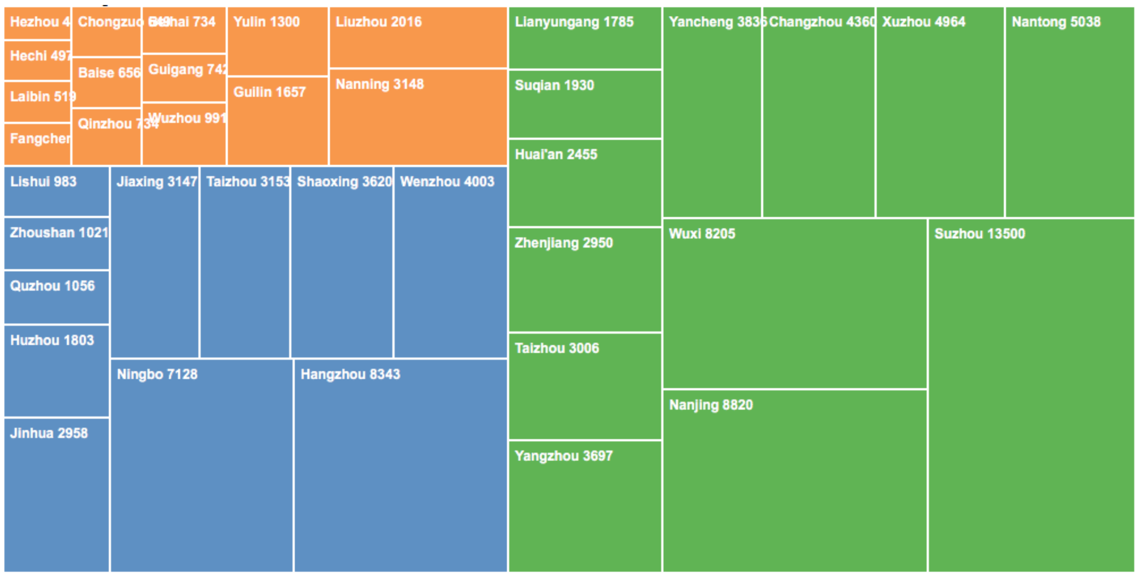


Figure 2. GDP of cities in three provinces in China, data gathered from Lv (2015) [30]

As can be seen in figure 2, this visualization makes the information easier to perceive by presenting numbers in the form of squares. The area size reflects the relative size of number, making it easy to compare between cities and provinces.

2.2.2 Scientific Visualization

Scientific visualization is a kind of visualization that focuses on multidimensional phenomena (architectural, meteorological, medical, biologic, etc.) primarily [2]. In other words, it is the visualization of data from scientific experiments and simulations. The data are measured or acquired via lengthy, expensive simulations. The most important aspects of scientific visualization are realistic renderings of volumes, surfaces and illumination sources. The content presented will be drawn with both visual design and information graphics prospects, and benefit from both [2]. Interactive visualization can be implemented to make the visualization more productive. When visualizing scientific data, the data often has missing values, various methods of handling the missing values such as interpolation can be applied. Figure 3 shows a 3D visualization of height data matrix of Maunga Whau of Auckland, New Zealand.

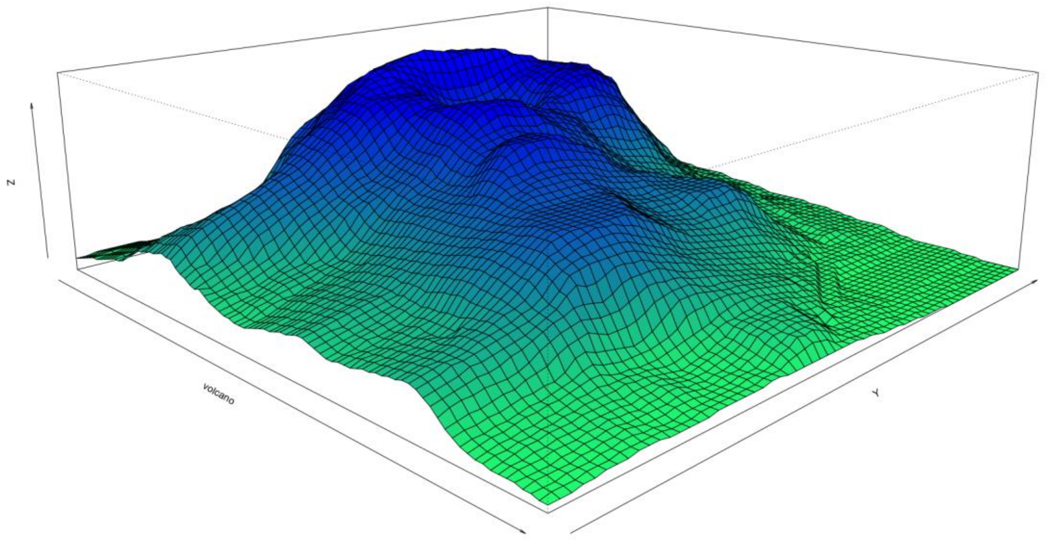


Figure 3. 3D Perspective view of Maunga Whau Volcano of Auckland, data gathered from R datasets Package

As shown in figure 3, the color of the surface changes gradually from green to blue as the height of the mountain increases. This figure also demonstrates that scientific visualizations benefit from visual design and information graphics, by using unrealistic color mapping to provide better perception of height while also remaining informative by correlating the height of points with color.

2.2.3 Mathematical Visualization

Mathematical visualization is visualization of mathematic problems. It is the visualization of data generated from mathematical equations using computer programs. The computation and the visualization both are very hard or nearly impossible to be achieved manually, and only became possible several decades ago, with the progress in computational and visualization capabilities. [5] A good example for mathematical visualization is the Mandelbrot set.

> Mandelbrot set is a set of complex c-values for which the orbit of 0 does not escape under iteration of $x^2 + c$. Equivalently, the Mandelbrot set is the set of c-values for which the filled Julia set of $x^2 + c$ is a connected set. [6]

A complex number belongs to the Mandelbrot set if, when starting with $z_0 = 0$ and applying the iteration of complex quadratic polynomial $z_{n+1} = z_n^2 + c$ repeatedly, the absolute value of z_n remains bounded however large n gets. Listing 1 shows the mathematical definition of Mandelbrot set.

$$z_{n+1} = z_n^2 + c$$
$$c \in M \Leftrightarrow \limsup_{n\to\infty} |z_{n+1}| \leq 2$$

Listing 1 Mandelbrot definition

As shown in listing 1, elements inside Mandelbrot set changes when n changes. As the complex numbers get more accurate and the iteration continues, the visual representation of the Mandelbrot set gradually demonstrates certain beautiful patterns that are very similar to patterns found in the nature. Figure 4 is the Mandelbrot set and an area called seahorse valley in 50 iterations.

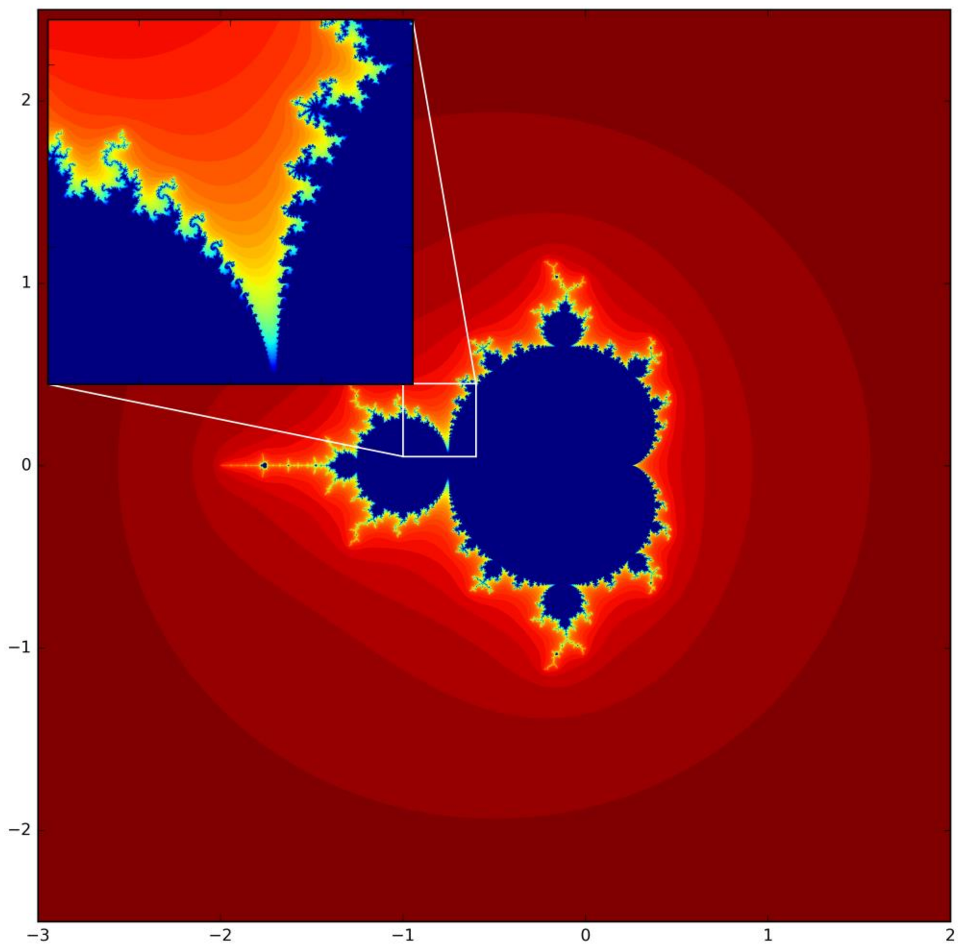


Figure 4 Mandelbrot set with zoom in view around seahorse valley

As shown in figure 4, each color in the graph represents the Mandelbrot set of one iteration. The biggest circle shape is the set of numbers after the first iteration. The inner most shape with deep blue color, which looks like a heart, is the collection of complex numbers that, after 50 iterations, are still within the boundary of the Mandelbrot set. The beauty of the Mandelbrot set is that whether the points that are in the Mandelbrot set and are close to the boundary in this iteration will remain in the set after next iteration seems arbitrary yet obeying certain rules. As the iteration increases, the visualization of Mandelbrot set becomes increasingly sophisticated, but if zoomed in, the patterns by the boundaries are composed of many simple patterns found in previous iterations. There are smaller bulbs attached to the biggest bulb, and smaller parts attached to the main body or the cardioid of the Mandelbrot set that look like the main body itself. This indicates that the behavior of the Mandelbrot set is very similar to things in nature, for example the relation between a tree trunk, branches and leaves.

The zoomed-in region in figure 4 is the area near the coordinate$(-0.75, 0.1i)$. The area is called seahorse valley because the shapes appearing in this region resemble the tail of a sea horse, and already it can be seen that the smaller sea horse tails look like the bigger ones. With enough processing power, the plot can be zoomed in infinitely. No matter how closely zoomed in, the boundary of the Mandelbrot set will always appear crinkly as if it was not zoomed in at all [7]. This demonstrates the complexity of the Mandelbrot set.

What the Mandelbrot set means is still an open question in mathematics, but after Benoit Mandelbrot first visualized the set of numbers in 1980, engineers and scientists have been inspired by it, designing antennas used in mobile phones today capable of receiving signals in different frequencies and rendering landscapes in movies and so on. [8]

2.2.4 Domain Specific Visualization

Different kinds of visualizations are used in different domains or fields. Business intelligence or customer survey results need to be presented in such a way that they help making business decisions. Geographic visualization uses data that is geographic by nature, and presents it in visual form to help the viewer better understand the data and support the analysis of geospatial data. Educational visualization is the visualization of some phenomena that is hard to understand by other methods. It aids the topic to be taught easier. [5]

There is also medical visualization, which has been widely used nowadays to help understand medical conditions and make treatment decisions. It is the art of transforming medical imaging data sets to 3D images with the help of computers. Figure 5 is an example of medical visualization of RNA and DNA.

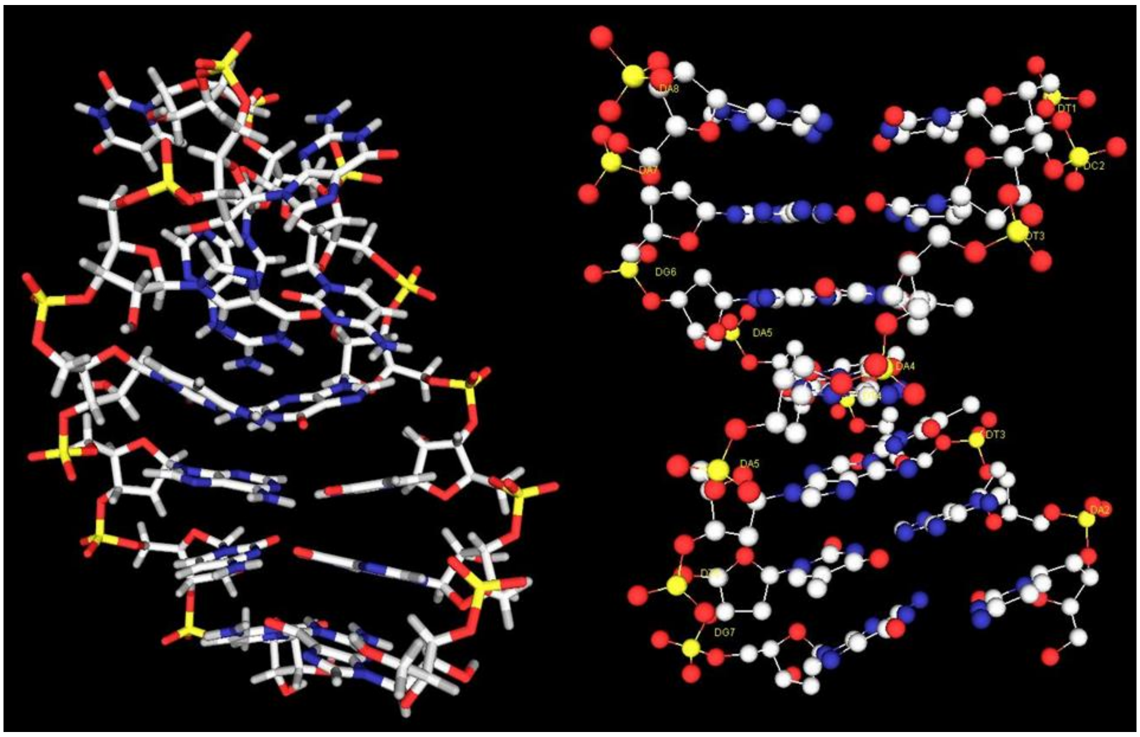


Figure 5 RNA and DNA visualization [26]

Figure 5 demonstrates one application of medical visualization, i.e. helping understand the structures of RNA and DNA. Though it is a relatively new frontier, medical visualization is developing rapidly thanks to the increase in computing power. Almost all cancer and surgery treatments rely on medical visualization in the developed world. [9]

2.3 Modes of Data Visualization

Interactive visualization is a mode of visualization used by a single viewer to explore and discover hidden meanings in data. When viewing such visualization, the viewer is granted full control of the data, and can change which data is being presented or how the data is presented. The quality of visualization will not be as polished as other modes because the visualization is generated in real-time. Google Public Data Explorer is a good example. [5]

Presentation visualization is the visualization used to deliver information to a wider group of audiences. This kind of visualization is often used in a video or presentation. The main difference between presentation visualization and interactive visualization is that, presentation visualization does not let the viewer to interact with the data, but just receive and observe the data being presented in the form that helps the viewer perceive. This kind of visualization is highly polished. [5]

Interactive storytelling is the mode of visualization that presents data through interactive web pages. The viewer cannot change the dataset but can alter the visualization to a certain degree, for example, change how the data is presented and change the data presented in the visualization. [5] Table 1 shows a comparison of the different modes of visualization.

Table 1 Comparison of different modes of visualization [5]

Visualization Mode	**User Interaction**	**Graphics Rendering**	**Target**	**Medium**
Interactive Visualization	controls over everything, including dataset	Real-time rendering	Individual or collaborators	Software or internet
Interactive Storytelling	filter or inspect details of preset datasets	Real-time rendering	Mass audience	Internet or kiosk
Presentation Visualization	User only observes	Precomputed rendering	Mass audience	Slide shows, video

As seen in table 1, different visualization modes are used in different scenarios and serve different purposes. This table can be used during design phase to help select visualization modes.

2.4 2-D Graphics

After selecting the mode for data visualization, the next step is understanding 2-D computer graphics that will be used for creating visualization. 2-D graphics are generations of digital images with computer. The following part will introduce raster graphics and vector graphics.

Vector Graphics are the graphics used for describing two-dimensional graphics with vectors. The points on a graph that the vector goes through are definite and possess attributes such as thickness, curve and shape. Graphics that require to be scaled will tend to use vector graphics. For example, architectural drawings, CAD programs, clip arts and fonts. Vector graphics have small file sizes, making them to load quickly and thus widely used in graphics on websites. Formats for storing vector graphics include PDF and SVG (Scalable Vector Graphics). [5]

Raster Graphics are graphics to display graphics on digital display such as computer screen and mobile screen. Raster graphics image has a matrix data structure, each cell of the matrix is the color data containing three values for color image, or one value for grayscale/binary image. On the screen, the pixels are assigned color to be displayed. The screen is scanned through and colored pixel by pixel, row by row. [10] The refresh rate denotes how many times the screen is being scanned every second. Most of the computer images are stored as raster graphic formats. Some of the formats include GIF, JPEG and PNG.

When a graph is generated through vector graphics, in order to be displayed on raster display, it needs to go through the process of rasterization. Figure 6 is a comparison of vector graphics and raster graphics.

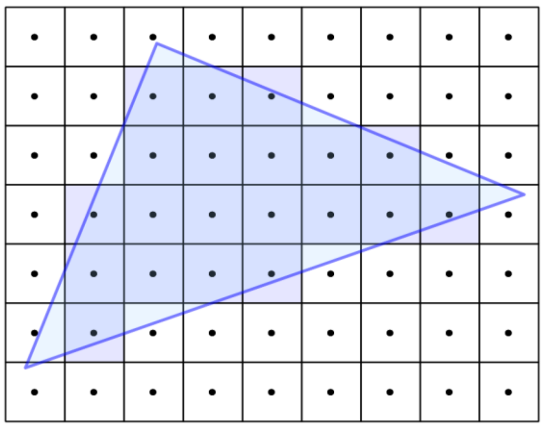


Figure 6 Comparison of Vector graphics and raster graphics [27]

As shown in figure 6, the vector graphics triangle is taken in and mapped on the pixel display system. Different algorithms are used to determine the illumination and the color of pixels. In figure 6, it is determined by whether the center of pixel is included in the inner side of the shape. After the process, the raster graphics display will only show a matrix of pixels with some being illuminated in light blue while the others remain white. After the process of rasterization, the lines of raster graphics will look like stairs on the edge, instead of straight lines or curves in vector graphics. This is called aliasing.

2.5 Data Formatting Libraries

Python is a high level general purpose programming language. It is used in many fields including software engineering, web backend servers and statistical programming. It is cross-platform and easy to learn, and it is also open source [11]. In this project Python was used, along with several Python libraries, for extracting the data from CSV and formatting the data to the proper shape accepted by D3.js and exporting the data in JSON (JavaScript Object Notation) format.

Pandas is a Python package that provides several data structures that are fast, expressible and flexible in addition to Python default data structures, to allow Python work with “relational” or “labeled” data easily and intuitively. It allows Python to perform practical data analysis. Its goal is to become the most powerful and flexible open source data analysis and manipulation tool in any language. [12] The most important use of Pandas library is the DataFrame data type, which provides an easy and fast way of tabular data I/O with many convenient functionalities for data manipulation.

2.6 Data Visualization Libraries

Matplotlib is a Python plotting library for the numerical library NumPy. It allows the user to write object-oriented codes for visualization, while also supporting MATLAB style procedural programming. It works with other libraries including NumPy for better performance. It provides high quality visualization meeting publication standard, accepts TeX document expressions and includes them in the output, can be implemented within applications and is easy to code and understand. [13] Besides, Matplotlib provides a wide range of control to the user so it is possible to optimize the visualization until it meets the demand. It has been the most popular Python visualization library and thus other libraries use it as default visualization library. The disadvantage is that it requires to be run in a Python environment, cannot provide web based interactivity and is not capable of making the planned visualization.

Bokeh is a Python visualization library that lets the user write the code in Python, and generates JavaScript output so that the plot result can be embedded into a website, while also provides interactivity [14]. The library is well documented and the functions are easy to use. At the cost of a bit less control than Matplotlib, it enables Python programmers to make web based interactive visualization. However, it still lacks the functionalities needed for making the planned visualization.

Chart.js is a JavaScript visualization library with minimalistic design philosophy and is very easy to use, thanks to good documentation. With a certain range of control over the visualization, it provides a quick approach to web based interactive visualization. However, it is a rather light-weight visualization library that can only make 8 types of charts such as line chart and bar chart. Chart.js still does not meet the functionality requirement for the project.

D3.js is a JavaScript library for creating and manipulating DOM documents with data. By working together with HTML and CSS, it gives a wide range of control over the visualization from the position of an individual data node to the font of text on visualization. It is a standardized library while is still open-source. [15] Being a JavaScript library, it provides a wide range of interactivity to the visualization. But it is very different from other visualization libraries in terms of generating of the visualization, and the documentation of D3.js is the worst compared to the other visualization libraries mentioned above, which makes it hard to learn and implement. However, it is the only visualization library that has the functionalities needed for the project and thus D3.js was chosen to be the data visualization library of this project. Version 3 D3 is used instead of the newer version 4, since there are more tutorials, books and examples available in version 3.

2.6.1 D3.js

D3 offers extensive control and is capable of making informative and elegant visualizations just as Matplotlib does, which is also a weakness. It requires the programmer to manually set up every element of the visualization from the ground up. For making “traditional” visualizations such as pie charts and line charts that other libraries can achieve with a few lines of code, D3 requires several dozen lines of code. For example, making a simple scatterplot, the difference is drastic. Listing 2 and 3 are the code snippets for generating scatterplot in D3 and Matplotlib respectively.

```
//width and height
var w = 500;
var h = 300;

var padding = 30;

//scale set up:
var xScale = d3.scale.linear()
  .domain([0, d3.max(dataset, function(d) {
    return d[0];
  })])
  .range([padding, w - padding * 2]);

var yScale = d3.scale.linear()
  .domain([0, d3.max(dataset, function(d) {
    return d[1];
  })])
  .range([h - padding, padding]);

var rScale = d3.scale.linear()
  .domain([0, d3.max(dataset, function(d) {
    return d[1];
  })])
  .range([2, 5]);


//define xAxis and yAxis

var xAxis = d3.svg.axis()
  .scale(xScale)
  .orient('bottom')
  .ticks(5);

var yAxis = d3.svg.axis()
  .scale(yScale)
  .orient('left')
  .ticks(5);

//create SVG element
var svg = d3.select('#scatterplot')
  .append('svg')
  .attr('width', w)
  .attr('height', h);

//create circles
svg.selectAll('circle')
  .data(dataset)
  .enter()
```

```
        .append('circle')
        .attr('cx', function(d) {
          return xScale(d[0]);
        })
        .attr('cy', function(d) {
          return yScale(d[1]);
        })
        .attr("r", function(d) {
          return rScale(10);
        });

      //create x and y axis
      svg.append('g')
        .attr('class', 'axis')
        .attr('transform', 'translate(0, ' + (h - padding) + ')')
        .call(xAxis);

      svg.append('g')
        .attr('class', 'axis')
        .attr('transform', 'translate(' + padding + ', 0)')
        .call(yAxis);
```

Listing 2. D3 scatterplot code

```
  plot(time, temp, 'o')
  Ylim([0, 75])
```

Listing 3. Matplotlib scatterplot code

Figure 7 presents the results of scatterplot of D3 and Matplotlib, with D3 visualization on the left and Matplotlib visualization on the right.

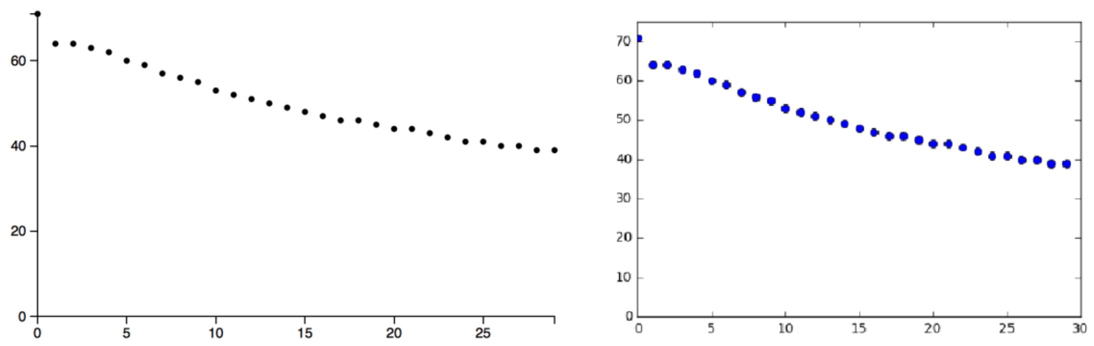


Figure 7 Comparison of D3 scatterplot (left), and Matplotlib scatterplot (right)

As figure 7 shows, the plot results are almost the same, while the code snippets from listing 2 and listing 3 pose a huge difference in the amounts of work. What the D3 code does is that it first sets up the canvas size, then based on the input domain and output range set up by the programmer, performs scaling on x values and y values on the points. Next, it defines the x axis and y axis. Afterwards, the plotting finally begins by selecting the SVG canvas, appending the circle elements to the data node on the canvas with coordinate data returned by scaling function. Finally, it adds the axis elements to the plot.

The plot is generated based on the data being bound to the elements, if the data is changed, the plot will be changed automatically, which is why D3 stands for data-driven documents. The D3 code example in listing 2 also demonstrates the wide range of controls given to the programmer to make the perfect plot in his or her mind. And since the plot is generated with JavaScript, it provides all kinds of web based interactivity, and can be regenerated on the fly within a web browser, while not being constrained by the environment like Matplotlib does.

Since D3.js version 1.0 was released in 2011, it gradually and steadily gained popularity among data visualization community. It is one of the most popular visualization libraries that there are already some libraries built on top of D3, as a “wrapper” to its functionalities, to provide easy and fast visualization solutions. However, the most important functionalities of D3 to the project are layouts.

2.6.2 D3 Layouts

A layout is a function or strategy for mapping data nodes on the canvas. It takes in data and calculates the positions for each element in the data and outputs new data that includes this information for plotting. [16] Figure 8 depicts the comparison of the plot generation process between D3 and other visualization tools.

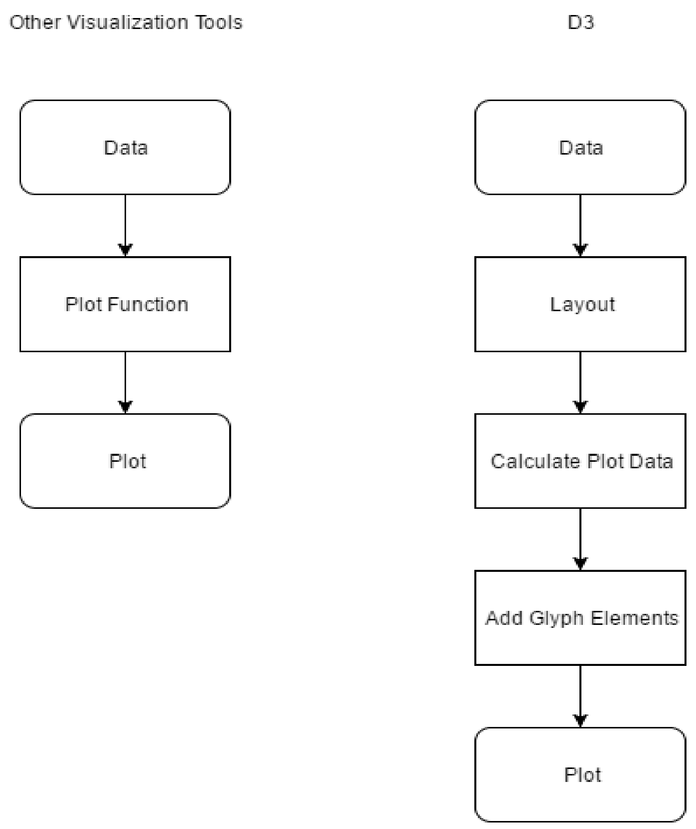


Figure 8. Comparison of plot generation process between D3 and other visualization tools (Matplotlib, Bokeh and Chart.js), data gathered from Zhihua Lv [17]

As figure 8 demonstrates, layouts are similar to plot functions from Matplotlib, Bokeh and Chart.js in terms of their roles in the process of visualization. The difference is that instead of drawing the visualization, D3 applies algorithms on imported data to generate information such as coordinate and hierarchy depth of the node for drawing. The other process such as binding data to SVG elements and actual drawing process still needs to be implemented afterwards. It is a rather low-level concept that other visualization tools do not have. It could be hard to comprehend at the beginning, but once the programmer are familiar with it, it is very convenient for making specific visualizations that the programmer has in mind. So for developing common and widely used visualization, Chart.js, Matplotlib and Bokeh are better options, while for making more sophisticated, more informative visualization or visualization with specific demand, D3 is the better option.

D3 provides 12 layouts including pack layout, partition layout, treemap layout, tree layout and cluster layout that are used to visualize hierarchical data and relations [17]. Hierarchical data are, for example family tree that contains parent-child relationships, with the youngest generation, presumably grandchildren at the lowest level, and the oldest generation, i.e. grandparents at the top; or academic publications with every publication being leaf nodes (nodes without children) at the lowest level, the writer at the level above the publication level, and the institute at the top. In this project, treemap layout and cluster layout are used, along with the bundle layout that is used to assist the cluster layout.

2.7 Analysis in the Mobile Game Industry

Mobile gaming has been growing rapidly in recent years, with thousands of new games being released on a daily basis. Many investments are being poured into the industry, and huge amounts of revenue are generated from the market. The global games market will reach $108.9 billion in 2017, with 42% coming from the mobile games market. [1] Figure 9 shows three major areas affecting the performance of a mobile game.

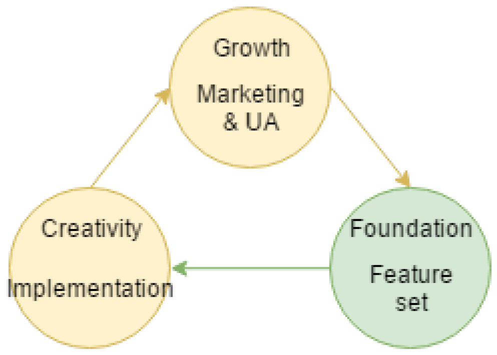


Figure 9 Three major factors affecting a mobile game's performance

As figure 9 demonstrates, Marketing & UA, implementation and feature set make up the most important factors of revenue potential of a game. The market trend is changing rapidly, and the industry is trying to understand the trend. There are many studies regarding mobile game monetization being conducted, but most of resources are being directed toward marketing, user acquisition, analytics, creativity IP/brand and overall concept of games. The mobile game industry lacks a fact-based decision-making and feature-selection system for game design. A study of over 7000 mobile game analyses in the past 3 years at GameRefinery found that the game design feature sets contribute 40-60% of the sustainable revenue of games.

This kind of feature level analysis requires huge amounts of resources and work. While AAA studios have the resource to conduct game breakdown analysis similar to what GameRefinery does at a very high expense, medium to small size studios have difficulty trying to keep up with the market trend. Many studios and developers make mobile game design decisions based on their personal preference and experience. What GameRefinery is trying to achieve is to create a platform for mobile game developers to better understand the market by looking at both high level generic trend of the market and detailed feature-level insights, and make fact-based feature-level game design decisions based on the analysis findings.

GameRefinery is a mobile game analytics company that conducts analysis to thousands of games. For each game, the company collects over 200 features to map the game ‘DNA’. After that the ‘DNA’ goes through the process of data crunching with machine learning algorithms and statistical methods. Then the findings are delivered to the clients in a well-formatted and easily-perceived way to help them improve and maintain game competence throughout the lifespan of their games and increase the monetization potential of the games.

As the company was developing its SaaS platform, better visualizations of the analysis data were considered. The data visualizations in the platform were traditional, unattractive and sometimes confusing. As the data and findings are very important to the company and its clients, it would be regretful if they were not well visualized, well communicated to and perceived by clients. Therefore, this project strives to explore new ways of visualizing game analysis data using new technologies with the goal of helping the client in mobile game design and feature selection.

3 Materials and Methods

This chapter starts by explaining reasons for selecting the visualizations and their design, then introduces cluster, bundle and treemap layouts used in the visualizations, followed by the process of data visualization, beginning from reading in database, data formatting and the most important aspects of generating these visualizations. Finally, the last part of the chapter describes the usability test.

3.1 Top 50 Game Genres Design - Cluster and Bundle Layout

As the company was developing the online platform for the clients and analysts, it missed visualization for presenting game genre trends. The company had genre data of all the games in the database, but lacked a good way of presenting them to the clients. When considering different visualizations for game genres, a visualization example on D3 official site stood out. It was a visualization of the Guardian news article titled “Violence and guns in best-selling video games” with a cluster layout and bundle layout visualization. The right part of the visualization was best-selling computer games, the upper left part was the content labels, and lower left part was the weapons in the games. The visualization linked the games with their content labels and weapons. Next to each name of game, name of weapon or content label there is a square with height correlated to the game sale, count of games that had that weapon or belong to that label, providing a straightforward representation of related data.

Unfortunately, the original article was removed, but the visualization greatly inspired the visualization of genre trends in this project. In GameRefinery analysis, a game was categorized into two genres, primary genre and secondary genre. The primary genre described the action layer, or combat layer of the game, while the secondary genre described the preparation layer, such as when the players were preparing for battle or developing their bases. For example, Clash of Clans would be categorized into the primary genre of strategy, and secondary genre of base development (hostile). So the original idea of the game genre trend visualization was, on the right part of the visualization was the top 50 games, with a square next to the name, representing the Game Power Score (GPS). GPS is a number ranging from 1 through 100, that benchmarks the game’s revenue potential. The width of square would represent the GPS. On the left part were the primary genre and secondary genres, and each genre was linked to all the games that belonged to it. A square would be added next to a genre, and its width would reflect the count of games in that specific genre. Cluster layout and bundle layout were used in this visualization.

3.1.1 Cluster Layout

The cluster layout produces dendrograms: node-link diagrams that place leaf nodes of the tree at the same depth. For example, a cluster layout can be used to organize software classes in a package hierarchy. [18] Figure 10 is a demonstration of selected cities and provinces in China, with tree layout on the left, and cluster layout on the right.

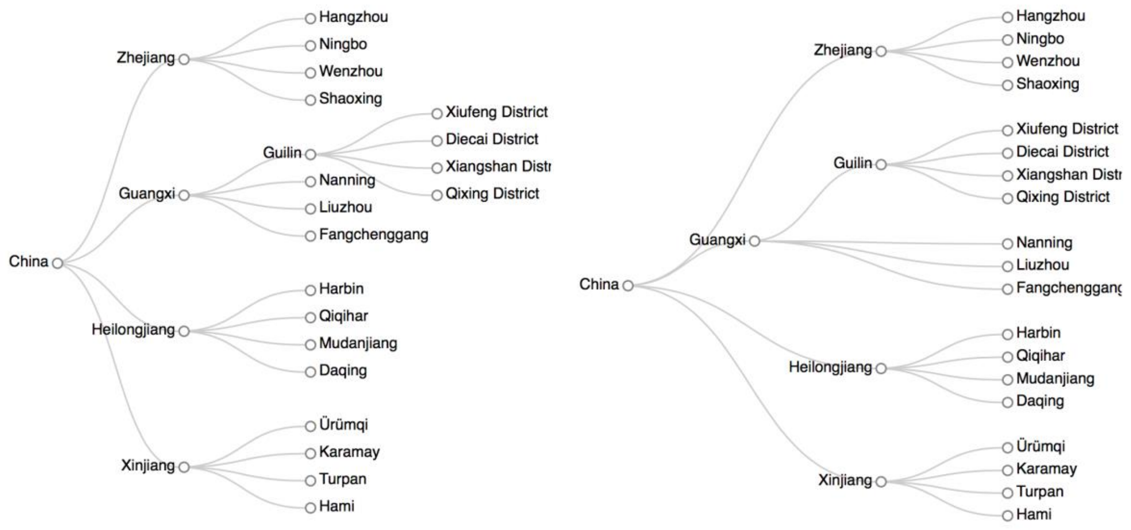


Figure 10 Comparison of tree layout(left) and cluster layout(right) visualization of selected cities in China, data gathered from [28]

As seen from the plot in figure 10, China on the left, is the highest level, and 4 provinces lie on the second level, cities that belong to a specific province are on the third level, and the fourth level shows 4 districts from the city of Guilin. The cluster layout arranges all the leaf nodes on the same lowest level, no matter how many levels are there between the leaf node and top node. That is, as seen from the plot on the right of figure 10, the lowest level includes cities from 4 different provinces and districts from the city of Guilin. All the leaf nodes will be on the same level, despite the fact that between different branches, there are different levels. The tree layout, which is very similar to the cluster layout, keeps each level on its own level graphically, as demonstrated by the plot on the left of figure 10. The cluster layout and tree layout are both great for representing hierarchical data.

Since cluster layout is used to represent hierarchical data, the form of data should be also hierarchical. One possible form of data is demonstrated by listing 4, a city hierarchy JSON data.

```
{
    "name":"United States",
    "children":[
        {
            "name":"California",
            "children":[
                {
                    "name":"Los Angeles"
                },
                {
                    "name":"San Francisco"
                }
            ]
        },
        {
            "name":"New York",
            "children":[
                {
                    "name":"New York"
                },
                {
                    "name":"Buffalo"
                },
                {
                    "name":"Rochester"
                }
            ]
        }
    ]
}
```

Listing 4 Cluster layout city data

Listing 4 demonstrates one possible JSON format data accepted by D3 cluster layout. The name of the object is “United States”, and its children property is a list of 2 state items, i.e. New York State and California. Within the “children” list of each state object, there are several selected cities. This object contains the hierarchical information. For this project, the data used in creating the cluster layout of top 50 game genre visualization was formatted similar to listing 4.

3.1.2 Bundle Layout

Bundle layout is a layout for generating links between different nodes. A path is generated through the input node upwards to the parent hierarchy, to the least common ancestor, then to the output node. This layout can be used as a function, together with other hierarchical layouts to generate bundled splines between nodes, as demonstrated in Figure 11. [19]

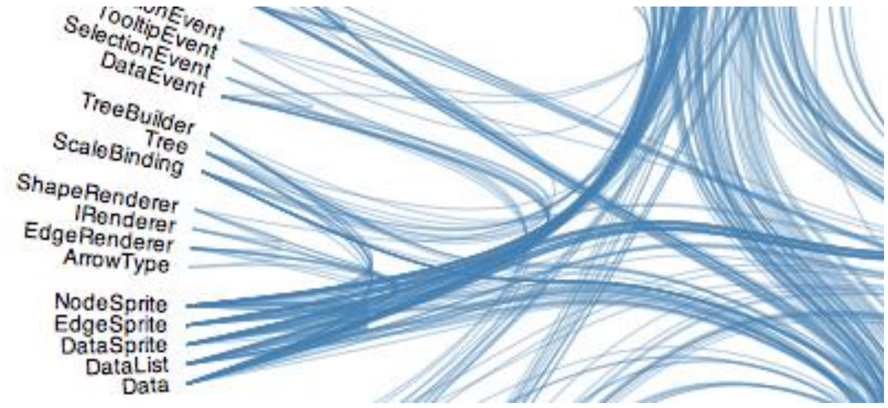


Figure 11. Visualization of software dependencies [19]

As seen in figure 11, bundle layout is used together with cluster layout to generate the visualization. The cluster layout decides the coordinates of nodes, and bundle layout generates the lines between the nodes. The bundle layout has one functionality: calculate the path from one node to another. Figure 12 is a representation of the relation links with straight and curved lines.

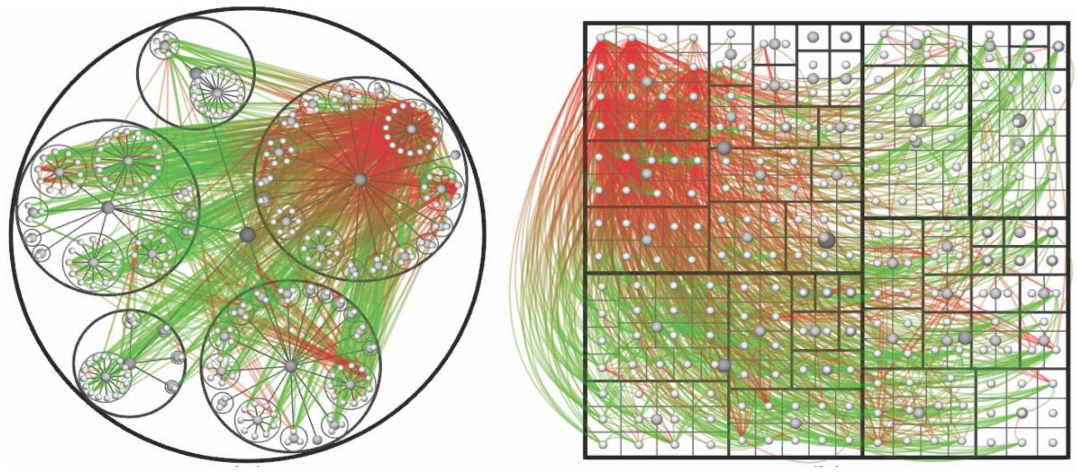


Figure 12. Adjacency representation with Straight lines and curved links [20]

As can be seen in figure 12, straight lines and curved links cannot represent adjacency between lines well, and thus cause visual clutter. While straight lines make the graph messy, curved lines make it hard to tell where each line is leading to. The D3 bundle layout used Danny Holten’s hierarchical edge bundling algorithm to tackle the problem by bundling the lines together [20]. It looks for the input nodes, then for bundling, it searches upwards to the parent nodes, until common ancestors between different lines are found. If some lines have common ancestors, their lines will be “bundled” closer together. This algorithm avoids problems presented by straight lines and curved links while also providing a very good hierarchical relation information. [20] Listing 5 shows a possible form of flight data for generating D3 links.

```
[
  {
    "source":"Los Angeles",
    "target":"San Francisco "
  },
  {
    "source":"New York",
    "target":"San Francisco"
  },
  {
    "source":"Buffalo",
    "target":"Los Angeles"
  },
  {
    "source":"Rochester",
    "target":"San Francisco"
  },
  {
    "source":"Los Angeles",
    "target":"New York"
  }
]
```
Listing 5. Airline flights between selected cities

Since the data used by bundle layout is for generating links or lines between nodes, each element in the data object will naturally describe the link relations between nodes. As listing 5 demonstrates, each element in the list is an object containing source and target properties, source being the start of the flight, target being the terminal. This data is artificially generated only for demonstration. The relations data used in the visualization is formatted similar to listing 5.

3.2 Feature Breakdown Design – Treemap

The mobile game analysis consisted of hundreds of features that were grouped into feature categories. Each individual feature and feature category had its own quantified contribution to the game's revenue potential. To help clients find out which features are the most important to be implemented in their game and prioritize their task, it was very important to let them acquire such information with convenience. Within GameRefinery SaaS platform there was one radial treemap feature breakdown visualization as shown in Figure 13.

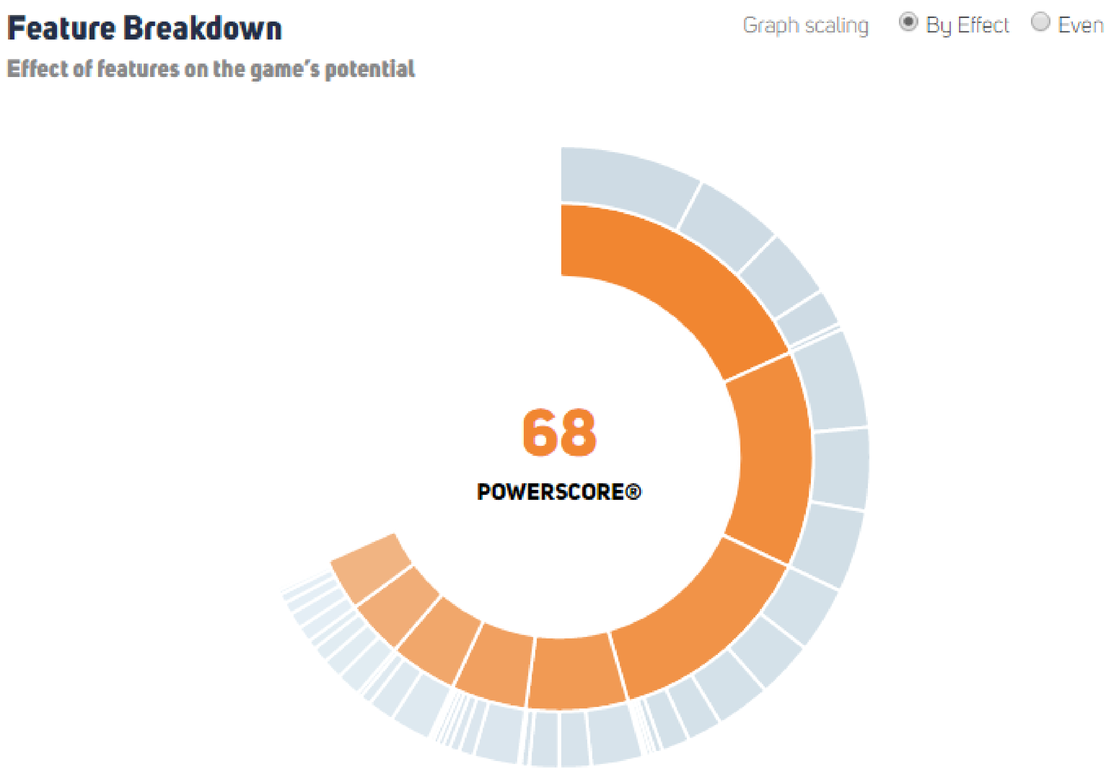


Figure 13. GameRefinery feature breakdown

As seen in figure 13, the visualization above is a radial treemap. Each element in the inner circle represents feature categories, and the elements in the outer circle represent individual features of that category. When hovered on the partition, the graph displays a tooltip showing the feature. The visualization was considered confusing, and was planned to be removed. However, treemap was actually good for presenting quantitative hierarchical data, and it was considered that a normal square treemap implementation would be more suitable for the feature breakdown visualization. The original plan was to make a treemap that groups features by their categories, and differentiate feature categories with different colors. The size of the square and the darkness of the color reflects their importance. If the feature is more important, the size of the square would be bigger and the color would be darker.

3.2.1 Treemap Layout

Treemap is a visualization of hierarchical structures in a limited space, it makes 100% use of space [21]. It represents attributes of leaf nodes effectively with size and color difference. With treemap, users can easily compare nodes and sub-trees of the tree from different scopes, and spot patterns and exceptions within the hierarchical structure. [22]

Treemap was designed by Ben Shneiderman during the 1990s, and his original design was nesting rectangles. Then for better comprehension, borders were added to the design to show the nesting and hierarchy. The size of each square represented the quantitative attributes, such as the size of a file in a hard drive. The hue of its color can represent categorical or hierarchical attribute, files of same format can have the same hue. The lightness difference can also represent the information regarding the file size, for example the bigger the file, the darker the color. [22] Figure 14 shows a treemap implementation to support large data sets of one million items.

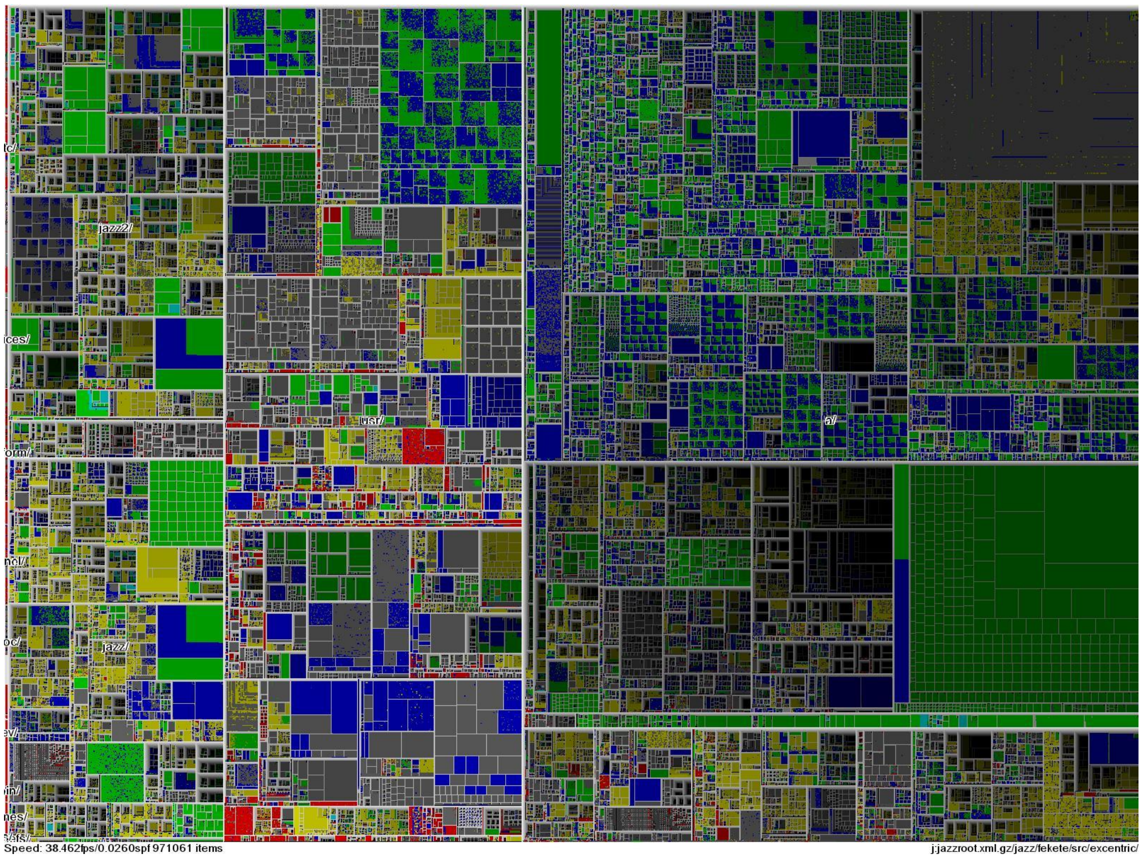


Figure 14. One Million Items Treemap [29]

Treemap visualization can be implemented in different ways. The normal treemap, as shown in figure 14, represents the overall structure and presents a good overview of the system, but if the data has many hierarchical levels, some low-level information might be missing. One way to solve the problem is to have a zoomable treemap. A zoomable treemap is implemented in environments such as web browser in such a way that, on the first view, it only presents the highest level and second level hierarchy node. By clicking any square, the selected node expands to cover the whole canvas and shows all the children nodes within the selected node. This way of implementation provides a high level structural overview while also maintaining all the low-level details within the information structure. This implementation is suitable for hierarchical data of multiple levels. Listing 6 shows GDP data for generating treemap visualization in D3.

```
{
  "name":"United States",
  "children":[
    {
      "name":"New York",
      "children":[
        {
          "name":"New York",
          "GDP":1234
        },
        {
          "name":"Buffalo",
          "GDP":5678
        },
        {
          "name":"Rochester",
          "GDP":9012
        }
      ]
    }
  ]
}
```

Listing 6. GDP data from selected US cities

Listing 6 showed one possible form of data accepted by the D3 treemap layout. All the data in listing 6 is randomly generated only for demonstration. For the object “United States”, it contains name property, and a “Children” property, i.e. a list of state objects. The state object in listing 6 is New York State, containing name and children property. The children property contains a list of city objects, with name and GDP property. In treemap, the value property (being GDP in listing 6) associated with the lowest level data is most important, because all the attributes associated with higher level objects used for generating the treemap plotting data are calculated based on their lowest level children’s values and hierarchy information. The data generated for the feature breakdown treemap visualization was similar to the data format shown in listing 6.

3.3 Reading Database

3.3.1 Database - Game Analysis

The database is the mobile game analysis database of GameRefinery. The database consists of several hundred pieces of analysis, with each piece of game analysis tracking over 200 features. The database used in this project was a CSV database. One feature example:

- How many soft currencies are there: The options for the feature are objective and quantitative. The analysts can acquire such information by playing the game and counting the numbers by themselves or searching for information on the internet.

Analysts analyze the games on the company SaaS internal tool. With several teams of analysts working on different markets, we manage to keep the data up-to-date and reveal the trend and pattern in the fast-changing mobile game market. These data were used for the data visualization.

3.3.2 Reading Data with Python Pandas Library

The database is formatted as a CSV file, with the first column being the features list, and the second column being the options of the list. The first row is the list of game names. Each column is a record of a game, with each cell in the column representing a feature with a specific option, initialized to be 0, and the selected options marked as 1. Listing 7 shows data input from csv.

```
df = pandas.DataFrame.from_csv('dataSet.csv', 0)
```

Listing 7. Read in dataset as pandas DataFrame

Listing 7 demonstrates the code for reading in the CSV data and assigning it to a Pandas DataFrame variable named `df`. After the operation above, the dataset was imported into the Python environment and is ready for formatting with Pandas.

3.4 Data Formatting

Now that the dataset is imported to the python environment, the next step is to format the data to the correct shape accepted by the D3 data visualization library. This chapter documents the process of formatting and outputting data for cluster, bundle and treemap layout visualizations.

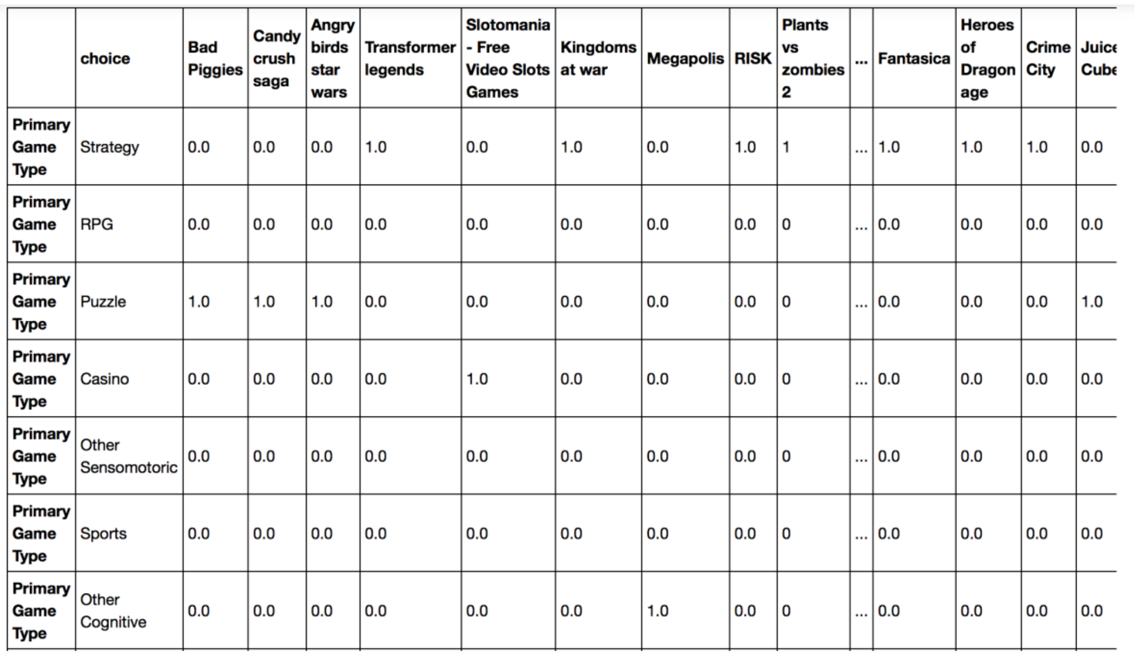
Since the database is not measured or generated by machine but gathered by analysts, they are well formatted with no empty cells or missing data. Thus, the work only requires formatting the data to the proper shape accepted by D3. Data formats accepted by D3.js include: JS objects, text files, JSON BLOB, HTML document fragments, XML document fragments, CSV files, and TSV (tab-separated values) files [23]. Within this project, Python Pandas and JSON library are used for data selection and output in JSON format.

3.4.1 Top 50 Game Genres - Cluster Layout and Bundle Layout

For this visualization, the data needed from the CSV were the primary genre types, secondary genre types and game names. First, extract the rows related to primary and secondary genres of the games, are shown in Listing 8.

```
primary = df.loc['Primary Game Type ', :]
secondary = df.loc['Secondary Game Type ', :]
```
Listing 8. Extracting primary and secondary genre

Listing 8 demonstrates the extraction of game genre related feature rows. The `.loc()` method is used by `dataframe` for label-location based indexer for selection by label. The code selects all the rows with indexer being ‘Primary Game Type’ and ‘Secondary Game Type. Figure 15 shows an overview of the primary subset.



	choice	Bad Piggies	Candy crush saga	Angry birds star wars	Transformer legends	Slotomania - Free Video Slots Games	Kingdoms at war	Megapolis	RISK	Plants vs zombies 2	...	Fantasica	Heroes of Dragon age	Crime City	Juice Cube
Primary Game Type	Strategy	0.0	0.0	0.0	1.0	0.0	1.0	0.0	1.0	1	...	1.0	1.0	1.0	0.0
Primary Game Type	RPG	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0	...	0.0	0.0	0.0	0.0
Primary Game Type	Puzzle	1.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0	...	0.0	0.0	0.0	1.0
Primary Game Type	Casino	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0	...	0.0	0.0	0.0	0.0
Primary Game Type	Other Sensomotoric	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0	...	0.0	0.0	0.0	0.0
Primary Game Type	Sports	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0	...	0.0	0.0	0.0	0.0
Primary Game Type	Other Cognitive	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0	...	0.0	0.0	0.0	0.0

Figure 15. Overview of primary subset

As seen in figure 15, cells in the choice column contain all primary game types(primary genre types). Cells in the first row represent a particular game, and each game’s column contains the information of primary game type. If a game belongs to a particular game type, the corresponding cell in that column will be marked 1.0. For example, Kingdoms at War is a strategy game and thus the cell representing strategy game type is marked with 1.0. Since generating `links` data between nodes takes less effort than generating `nodes` data and the `links` data can be used later for generating hierarchical data, `links` data from the `DataFrame` were generated first. It began by creating a `relations` list, then looping through each cell in every game column, if the value in the cell was 1.0, the column name of the cell, which was also the game name, would be returned as the source. The indexer of row that the cell was in, would be returned as the target. The source and target would together make up a Python dictionary, which would be appended into the `relations` list.

Secondly, creating `nodes` object starting by extracting the first 50 games from `df` DataFrame. A `nodesDict` was created to contain all the nodes, giving it an empty name value, and an empty `children` list. The nodes would be games, primary and secondary genres. A for loop then looped through the column names of `df` DataFrame, which were also game names. For each loop, a dictionary as an element of the `children` list of `nodesDict` was created, with name attribute being the game name, category attribute to be "game", and GPS attribute being a randomly generated GPS value, since the database did not contain the GPS data.

The next step was then initializing a new Python dictionary called `categoryDict`. This dictionary would be used later for recording games count of every genre. Then looping through the `primary` and `secondary` DataFrame genres, and appending them to the `nodesDict` with name being the genre name, category being either "primary" or "secondary" depending on which DataFrame they were from, and initializing `categryDict`, with key being genre name, and value being 0.

So far the `nodesDict` contained all the games with their `name`s, `GPS`s and `categories`, also genres with their `name`s and `category`s, but missing the genre game count. Thus, the next step was looping through `relations`, and using the `target` attribute of each element of `relations` list as a key, and incrementing the value of the key inside `categoryDict`. After that the `categoryDict` would contain the genre name and genre game count. Next, another loop would go through the `catego-ryDict` and `nodesDict`, and append the value of keys inside `categoryDict` to `gamesCount` attribute of genre elements inside `children` attribute of `nodesDict`. Finally the python objects would be exported and stored as two JSON file: `nodes.json` and `reltions.json` with `json.dump()` function. The complete code can be found in appendix 1.

3.4.2 Treemap Layout

The dataset used for treemap was almost the same, except that the dataset contained the feature category as an index of the dataset, and feature names were in the second column, with third column being feature choices, fourth column being the choice or option of the game from analysis, and the last column being the weight of the feature choice. The weights of all the features presented in the game made up the game GPS. The selected analysis for implementing visualization was an analysis of Candy Crush Saga.

The process of generating the treemap visualization JSON data began with creating an `index` list, containing all the indexes (which were also the feature categories) that were extracted from the CSV. If the index label did not exist in the `index` list, the index label would be added to the list. After that, a new Python dictionary called `data` was created, with its name being Candy Crush Saga, and a `children` attribute initialized to be an empty list. Then all the elements inside `index` list were appended to the `children` attribute of the `data` dictionary, along with an empty `children` list. So far, the game `data` dictionary contained a `name`, a `children` list containing all the feature genre elements. For each feature genre element, there was the `name` and an empty `children` list.

The next step was adding features and options that were of a specific feature category into those empty `children` lists. It was a loop going through the game analysis looking for value 1, if the value was one, it would check the game category, and find the category within the game `data` dictionary, and append the feature, the choice and the value to the `children` list. Finally, JSON library was used to convert the python object to a JSON object and output that JSON object to a file called `treemapNodes.json`. The complete code can be found in Appendix 2.

The data formatting and outputting of the treemap layout was relatively easy in comparison to that of the cluster and bundle layout, but the idea was the same, i.e. making the data into the correct shape accepted by the D3 layouts.

3.5 Plotting

After formatting the data, the next step was writing the code for plotting with D3. Since D3 is a JavaScript library, the plotting started by setting up a webpage with basic elements and a div element, also including the D3 library source to the page. Next a new `script` section tag was created and the visualization code would be written inside the tag.

3.5.1 Top 50 Game Genres

The function `d3.json()` is for reading external data into the webpage. For top 50 game genres, two nested d3.json() functions were used to read in both `nodes.json` and `relations.json`, as demonstrated by Listing 9.

```
d3.json('nodes.json', function(games) {
  d3.json('relations.json', function(relations) {})
})
```

Listing 9. Reading in external `nodes.json` and `relations.json` as games and relations object

As seen in listing 9, nodes data were first imported with relations data right after it. Then for sorting the nodes, the game nodes were sorted based on their GPS and game nodes with higher GPS came first. For primary and secondary game genres, they were sorted based on the games count. Next, the width and height of the SVG element were set, along with the radius of the radial cluster layout. Then for better graphics, two gradients for links from games to primary and secondary genres were set up, which later turned out to be confusing, and will be explained in the result chapter. But for the gradient, it was planned that the color of games is blue, primary genres are green and secondary genres are red. The two gradient coloring rules for the links were set to variables `primaryGradient` and `secondaryGradient`. The next step was setting cluster layout and generating `nodes` correlated to the layout. Listing 10 shows the setup of the cluster layout and `nodes` generation.

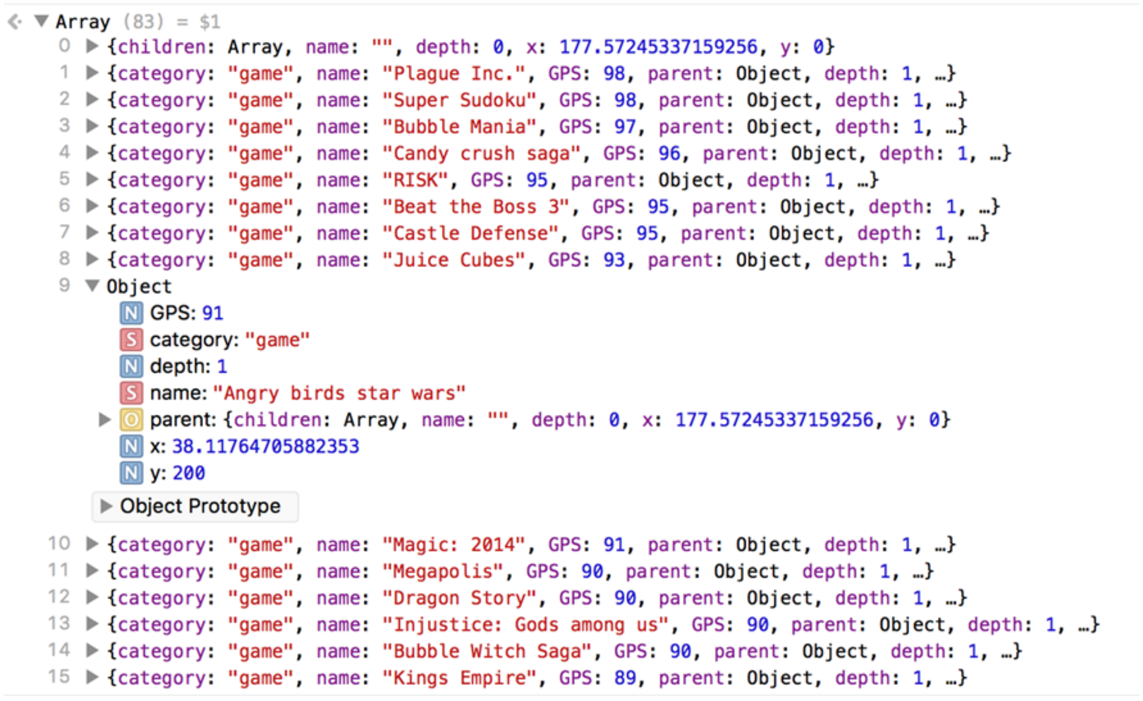
```
var cluster = d3.layout.cluster()
  .size([360, r])
  .separation(function(a, b) {
    return (a.category == b.category ? 1 : 2) / a.depth;
  });
var nodes = cluster.nodes(games);
var bundle = d3.layout.bundle();
var nodeWidth = (r * 2 * Math.PI / nodes.length) - 6;
var oLinks = map(nodes, relations);
var links = bundle(oLinks);

function map(nodes, relations) {
  var hash = [];
  var resultLinks = [];
  for (var i = 0; i < nodes.length; i++) {
    hash[nodes[i].name] = nodes[i];
  }
  for (var i = 0; i < relations.length; i++) {
    resultLinks.push({
      source: hash[relations[i].source],
      target: hash[relations[i].target],
    });
  }
  return resultLinks;
}

var line = d3.svg.line.radial()
  .interpolate('bundle')
  .tension(0.85)
  .radius(function(d) {
    return d.y;
  })
  .angle(function(d) {
    return d.x / 180 * Math.PI;
  });
```

Listing 10 Cluster and bundle layout initialization, nodes and links generation

Listing 10 demonstrated the cluster and bundle layout initialization and `nodes`, `links` generation. The `size()` method set up the layout to be radial, with a range of 360 degrees, and radius being variable `r` predefined. The `separation()` method separated adjacent nodes of the same category by 1, and nodes of different category by 2. This way it grouped the game nodes, primary nodes and secondary nodes together while separating the nodes of different categories. The `nodes` variable would be the fundament of drawing nodes onto the SVG. Figure 16 shows the `nodes` object after D3 cluster function.

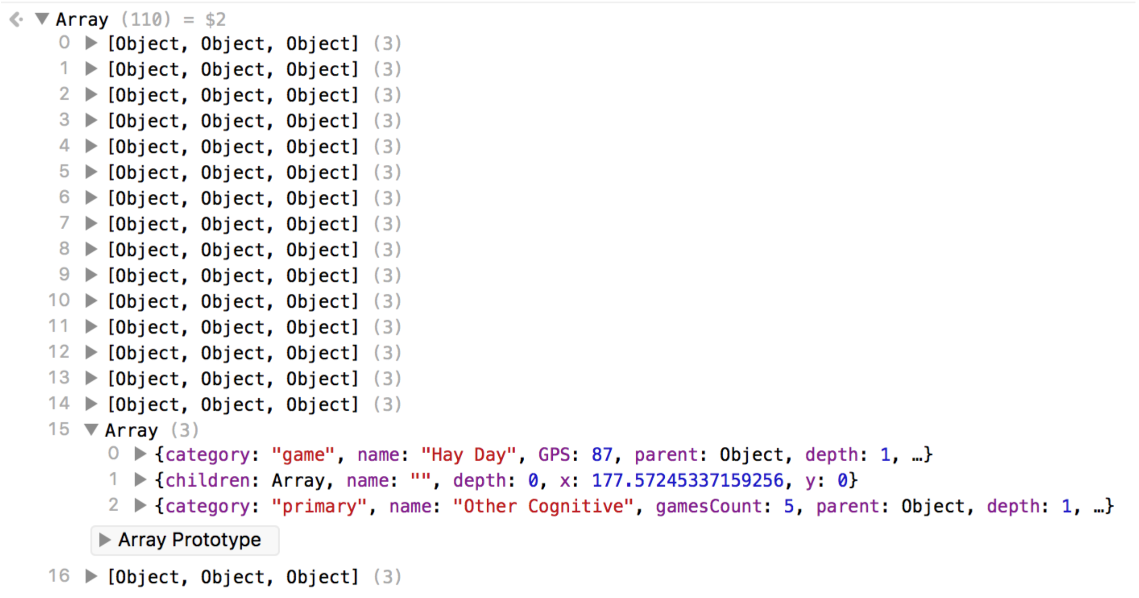


```
<· ▼ Array (83) = $1
  0 ▶ {children: Array, name: "", depth: 0, x: 177.57245337159256, y: 0}
  1 ▶ {category: "game", name: "Plague Inc.", GPS: 98, parent: Object, depth: 1, …}
  2 ▶ {category: "game", name: "Super Sudoku", GPS: 98, parent: Object, depth: 1, …}
  3 ▶ {category: "game", name: "Bubble Mania", GPS: 97, parent: Object, depth: 1, …}
  4 ▶ {category: "game", name: "Candy crush saga", GPS: 96, parent: Object, depth: 1, …}
  5 ▶ {category: "game", name: "RISK", GPS: 95, parent: Object, depth: 1, …}
  6 ▶ {category: "game", name: "Beat the Boss 3", GPS: 95, parent: Object, depth: 1, …}
  7 ▶ {category: "game", name: "Castle Defense", GPS: 95, parent: Object, depth: 1, …}
  8 ▶ {category: "game", name: "Juice Cubes", GPS: 93, parent: Object, depth: 1, …}
  9 ▼ Object
        N GPS: 91
        S category: "game"
        N depth: 1
        S name: "Angry birds star wars"
      ▶ O parent: {children: Array, name: "", depth: 0, x: 177.57245337159256, y: 0}
        N x: 38.11764705882353
        N y: 200
      ▶ Object Prototype
 10 ▶ {category: "game", name: "Magic: 2014", GPS: 91, parent: Object, depth: 1, …}
 11 ▶ {category: "game", name: "Megapolis", GPS: 90, parent: Object, depth: 1, …}
 12 ▶ {category: "game", name: "Dragon Story", GPS: 90, parent: Object, depth: 1, …}
 13 ▶ {category: "game", name: "Injustice: Gods among us", GPS: 90, parent: Object, depth: 1, …}
 14 ▶ {category: "game", name: "Bubble Witch Saga", GPS: 90, parent: Object, depth: 1, …}
 15 ▶ {category: "game", name: "Kings Empire", GPS: 89, parent: Object, depth: 1, …}
```

Figure 16. Nodes object

Figure 16 shows the `nodes` object and its contents. For each element of the `nodes` object, it contained the `name`, the `category` and `GPS` generated inside the `nodes.json`, and other values including the `parent`, `x` and `y` values generated with the `cluster()` function. The `parent` was the parent of the node, `x` and `y` were the coordinates where it should be drawn on the SVG. In the case of radial cluster layout, `x` would be the degree, `y` would be the radius.

After `nodes` was generated, the `map()` function was created for mapping the links between nodes, with the data from `relations` and `nodes` object. By looping through `relations`, it converted elements in `relations` from strings to objects that D3 recognizes. Figure 17 shows what the `links` object looked like after D3 bundle function.



```
▼ Array (110) = $2
  0  ▶ [Object, Object, Object] (3)
  1  ▶ [Object, Object, Object] (3)
  2  ▶ [Object, Object, Object] (3)
  3  ▶ [Object, Object, Object] (3)
  4  ▶ [Object, Object, Object] (3)
  5  ▶ [Object, Object, Object] (3)
  6  ▶ [Object, Object, Object] (3)
  7  ▶ [Object, Object, Object] (3)
  8  ▶ [Object, Object, Object] (3)
  9  ▶ [Object, Object, Object] (3)
  10 ▶ [Object, Object, Object] (3)
  11 ▶ [Object, Object, Object] (3)
  12 ▶ [Object, Object, Object] (3)
  13 ▶ [Object, Object, Object] (3)
  14 ▶ [Object, Object, Object] (3)
  15 ▼ Array (3)
       0 ▶ {category: "game", name: "Hay Day", GPS: 87, parent: Object, depth: 1, …}
       1 ▶ {children: Array, name: "", depth: 0, x: 177.57245337159256, y: 0}
       2 ▶ {category: "primary", name: "Other Cognitive", gamesCount: 5, parent: Object, depth: 1, …}
      ▶ Array Prototype
  16 ▶ [Object, Object, Object] (3)
```

Figure 17 Links object

As seen in figure 17, each element within the array was one link, and each link was an array of three elements. The three elements together described the source, the target, and their common ancestor of the link. The common ancestor in this case was the highest-level object with no name. After the function, the next step was defining the `line` function for drawing the links. The `tension()` method set up how tightly lines of same ancestor should be bundled together, the `radius()` set the line function to be drawn within the radius, and the `angle()` was the angle of the source and target of line, converted from degree to radian.

After the set up described above, the drawing functions and objects needed were ready, and it was time to plot the visualization. The following will explain the rectangle generation for nodes, which was a very important part of the visualization. Listing 11 shows the code for generating node rectangles.

```
var gBundle = svg.append('g')
  .attr('transform', 'translate(' + (width / 2) + ', ' +
(height / 2) + ')');

var node = gBundle.selectAll('.node')
  .data(nodes.filter(function(d) {
    return !d.children;
  }))
  .enter()
  .append('g')
  .attr('class', 'node')
  .attr('transform', function(d) {
    return 'rotate(' + (d.x - 90) + ')trans-late(' + d.y + ')'
+ 'rotate(' + (90 - d.x) + ')';
  });

node.append('rect')
  .attr('class', 'rectangles')
  .attr('width', nodeWidth)
  .attr('height', function(d) {
    if (d.GPS) {
      return d.GPS / 5;
    } else if (d.gamesCount) {
      return d.gamesCount * 2;
    } else {
      return 0;
    }
  })
  .style('fill', function(d) {
    if (d.GPS) {
      return 'hsla(240, 100%, ' + (144 - d.GPS) + '%, 0.9)';
    } else if (d.category == 'primary') {
      return 'hsla(120, 100%, ' + (54 - d.gamesCount * 2) +
'%, 0.9)';
    } else {
      return 'hsla(0, 100%, ' + (54 - d.gamesCount * 2) + '%,
0.9)';
    }
  })
  .attr('transform', function(d) {
    return 'rotate(' + (d.x + 180) + ')trans-late(-' + node -
Width / 2 + ', 2)';
  });
```

Listing 11 Node rectangle generation

As seen from listing 11, the process started by creating a `node` variable, and from the SVG `gBundle` element, selecting all `node` classes, binding data being elements inside the `nodes` variable excluding node elements with not children, and assign them the class of node. Since the radial cluster layout started from positive x axis, the whole layout was transformed to rotate 90 degrees anti-clockwise, translated the nodes to be of the distance of radius from the center, then rotate them back to pointing outward perpendicular to the surface of the circle.

The following step was appending a rectangle for each node, and setting its heights to be correlated to their values. For games, their height would be their `GPS` value divided by 5, for game genre nodes, their height would be `gamesCount` value times 2. For the filling of rectangles, `hsla()` color was used. Every node of the same category would have the same hue and saturation, but the lightness was generated based on its value.

For game nodes, the higher the GPS, the lower the lightness, which is the same case with game genres, the higher the `gamesCount`, the lower the lightness. The `d.gamesCount * 2` made the lightness vary wider so it was clearer to tell the color difference.

After the above configuration, the most important part of the cluster visualization was done. The nodes of different categories were nicely bundled and their values were visually represented by the height of the rectangle, also graphically by the brightness of the color. Figure 18 is the final version of top 50 games genre visualization.

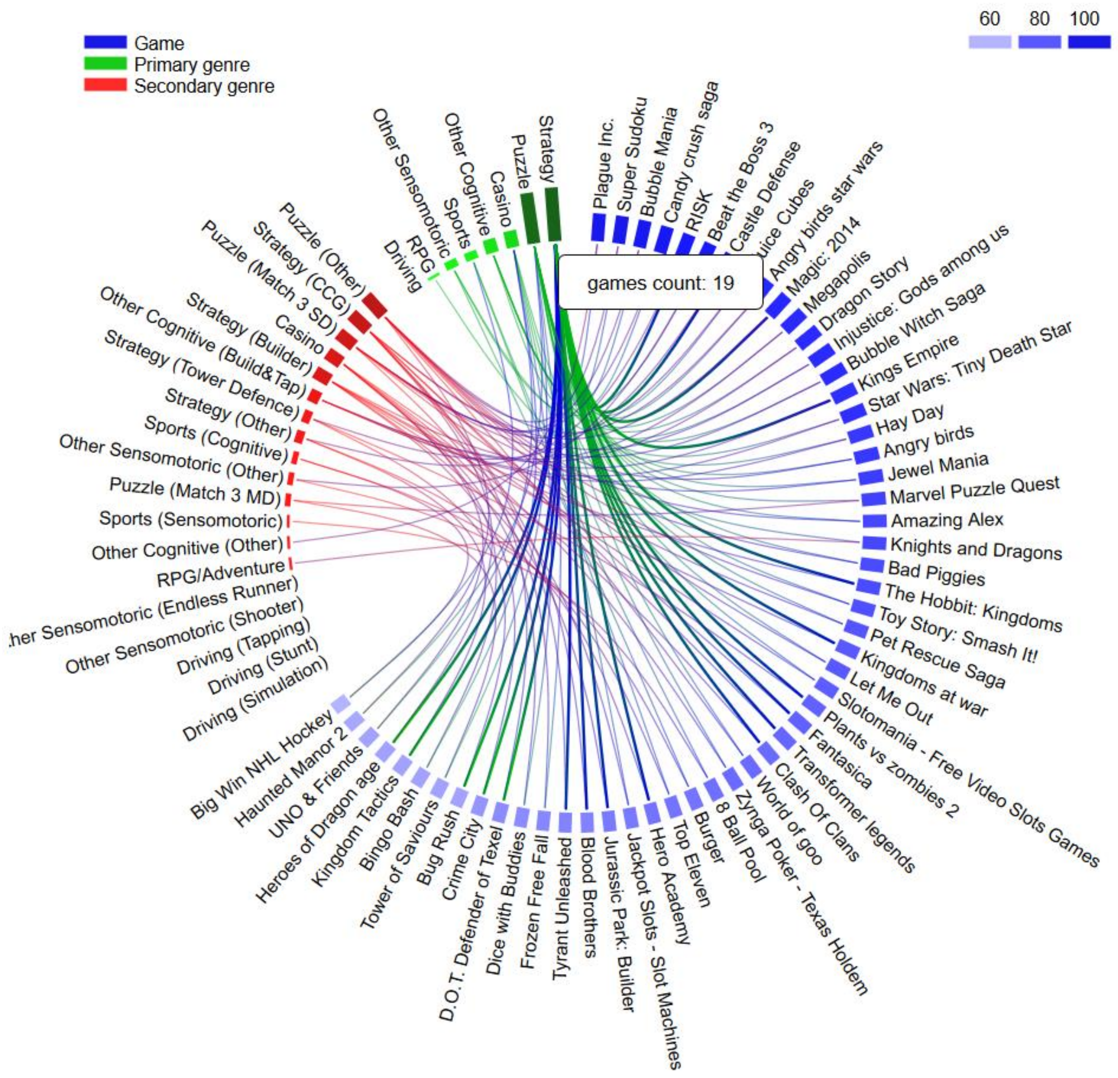


Figure 18 Top 50 games genre visualization

As shown in figure 18, the final visualization result of game genre trend presents a clear view of genre trend among popular games. The games and genres are sorted so it is clear to see which game or genre is more popular. When hovering on top of a genre node, the links will be highlighted and a box with the count of games of that genre will appear. After completing all the steps mentioned above, what remained to be done were drawing the links, adding the name text next to the nodes, adding tooltip function showing more information when hovering the mouse above it. Also, making the links related to the node stand out by increasing the stroke width, adding few explanatory tips on the graph, and setting CSS for the classes needed to be implemented. The complete code can be found in Appendix 3.

3.5.2 Feature Breakdown

The starting setup for the treemap layout was the same as cluster layout starting setup. After setting the SVG width and height, two layers `layer1` and `layer2` were appended to the SVG element for drawing the feature squares and feature genre text. Using two layers prevents the feature genre text from being shadowed by the game feature squares.

After reading in the `treemapNodes.json` and converting it to `nodes` objects with `treemap.nodes()` function, the following part of treemap visualization was setting an object tracking the domains of a feature genre based on the range of the feature weights, and making a scaling function for the lightness. Listing 12 demonstrates the creation of the domain object and scaling function.

```
var valueDict = {};
var categoryList = [];

for (i = 0; i < nodes.length; i++) {
  if (nodes[i].depth == 2) {
    if (nodes[i].parent.name in valueDict) {
      valueDict[nodes[i].parent.name].push(nodes[i].value);
    } else {
      valueDict[nodes[i].parent.name] = [nodes[i].value];
      categoryList.push(nodes[i].parent.name);
    }
  }
}

var domainDict = {};

for (i in categoryList) {
  domainDict[categoryList[i]] = [val -
ueDict[categoryList[i]][0], valueDict[categoryList[i]][0]];
```

```
    for (j = 1; j < valueDict[categoryList[i]].length; j++) {
      if (valueDict[categoryList[i]][j] < domain -
  Dict[categoryList[i]][0]) {
        domainDict[categoryList[i]][0] = val -
  ueDict[categoryList[i]][j];
      } else if (valueDict[categoryList[i]][j] > domain -
  Dict[categoryList[i]][1]) {
        domainDict[categoryList[i]][1] = val -
  ueDict[categoryList[i]][j];
      }
    }
  }

  function scaleLightness(domainElement, value) {
    if (domainElement[0] == domainElement[1]) {
      return 45;
    } else {
      return 100 - ((value - domainElement[0]) / (domainEle -
  ment[1] - domainElement[0]) / 2 + 0.20) * 100;
    }
  }
```
Listing 12. creating domain object and scaling function definition

As seen in listing 12, the data generation process started by creating a `valueDict` object and `categoryList` array. `ValueDict` contained an array of dictionary elements, and each dictionary was a key-value pair. In the case of the project, each dictionary had a key of feature category name, and the value of that dictionary was an array of all the feature values of that specific feature category. The data of the project was constructed in such way that all the feature genres belong to the first level, and all the features belong to the second level. The code looped through all the nodes that belong to the second level, and checked if the node’s parent attribute (which was the feature category name) was a key to a dictionary element in `valueDict`. If so, the node’s value was appended into that dictionary in the `valueDict`. If not, first a new dictionary with the key of the name of the parent would be initialized, and the node’s value was appended to the dictionary, then the dictionary was appended to the `valueDict`. Finally, the `cate-goryList` was initialized and the node’s parent name was appended into it.

The next step was creating the `domainDict` object. Within the `domainDict,` the code made an array of dictionaries with their keys to be the feature category name from the `categoryList`. Then it looped through all the elements inside `valueDict` to find the minimum and maximum values of a category, and assign them to the corresponding category elements of `domainDict`. So the result of `domainDict` would be an object containing an array. Each element in the array had a key of feature category name, and the value of the element would be 2 integers. The first is the minimum value, and the second is the maximum value of features in that specific feature category.

Finally the function `scaleLightness()` was defined, which took in a `domainElement`, array denoting the range of a category from the `domainDict`, and a value of a specific feature of that category. If in a feature category, there was only one feature, where scaling did not make much sense, it returned 45. If there was a domain, it first normalized the value into a new value within the range of 0 to 1. Since the weights of features varied in a wide range, they were shrunk down the range by the factor of 2, and adding 0.2 to make it closer to the center of 0.5, then multiplied by 100 and let 100 subtracted that value. Thus, after the scaling function the result was a number in range from 0 to 100, with higher weight returning a smaller lightness value, which resulted in darker brightness value. Nine colors were picked to represent 9 different feature categories. Listing 13 demonstrates initializing color and node binding.

```
var colors = ['hsla(0, 100%, ', 'hsla(210, 100%, ', 'hsla(120,
100%, ', 'hsla(337, 100%, ', 'hsla(300, 100%, ', 'hsla(180,
100%, ', 'hsla(24, 100%, ', 'hsla(80, 100%, ', 'hsla(60, 100%,
'];

var rectColors = {};
for (i in categoryList) {
  rectColors[categoryList[i]] = colors[i];
}

var node = layer1.selectAll('g')
  .data(nodes.filter(function(d) {
    return d.parent && d.value != 0;
  }))
  .enter()
  .append('g');

var genreNode = layer2.selectAll('g')
  .data(nodes.filter(function(d) {
    return d.parent && d.children;
  }))
  .enter()
  .append('g');

var rect = node.append('rect')
  .attr('class', 'rect')
  .attr('x', function(d) {
    return d.x;
  })
  .attr('y', function(d) {
    return d.y;
  })
```

```
    .attr('width', function(d) {
      return d.dx;
    })
    .attr('height', function(d) {
      return d.dy;
    })
    .style('fill', function(d) {
      if (!d.children) {
        var colStr = rectColors[d.parent.name] + scaleLight-
ness(domainDict[d.parent.name], d.value) + '%, 1)';
        return colStr;
      } else {
        return 'white';
      }
    });
```

Listing 13 Color initialization and node data binding of two layers

As seen from listing 13, the `colors` array was first created, then the color array was combined with category data to create `rectColors` object to associate the categories with colors. Then `node` was bound with feature data in `layer1` and `nodeGenre` was bound with feature category data. For the filling of each feature rectangles, a `hsla()` string from `rectColors` was constructed. The string is composed of fixed hue, saturation, a lightness generated through `scaleLightness()` function, and alpha being 1. The string was returned as the fill argument. This made it possible to generate colors with different lightness based on the individual weight of features. Figure 19 is the result of treemap feature breakdown visualization.

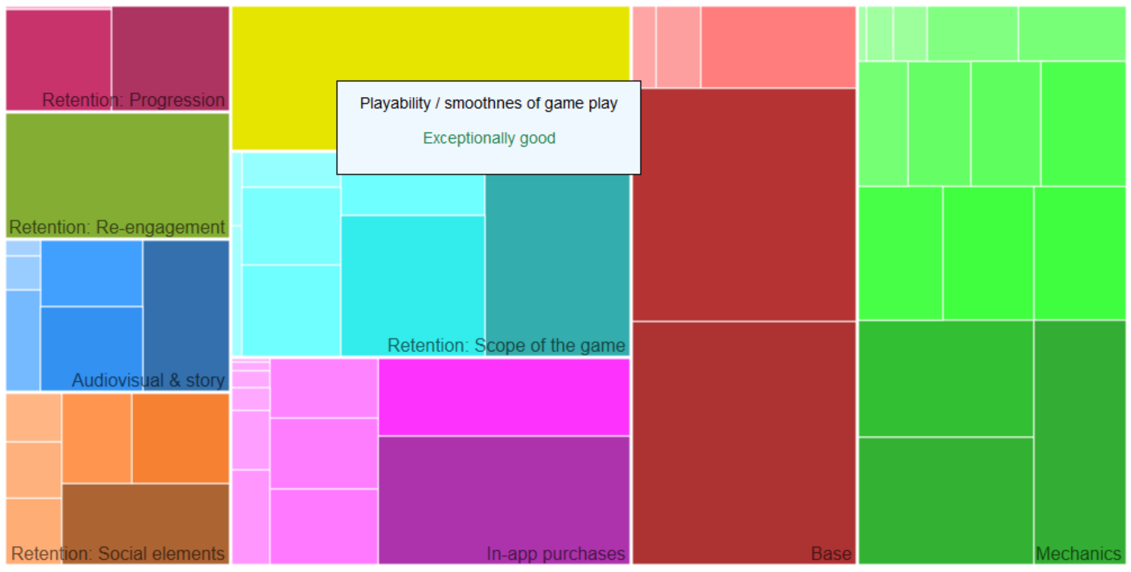


Figure 19 Feature breakdown visualization

Figure 19 is the final treemap visualization of the feature breakdown. After the previous steps mentioned above, the following steps were needed to make the visualization in figure 19: drawing the genre text on the `layer2`, adding mouse hover interactivities with a tooltip showing the feature name and the feature choice, and setting CSS for different classes. The complete code can be found in Appendix 4.

3.6 Usability Test

A usability test was conducted to evaluate the performance of visualizations. The usability test was made up of two parts. First, the participants could familiarize themselves with the visualizations by completing small tasks, then the participants would fill in a SUS (System Usability Scale) form, and answer some open questions. The SUS is generally used for evaluation of a system, after the participants have tested the system. It is a fast and easy method for effective evaluation of usability of a system with small sample sizes. [24] The participants were five analysts from GameRefiney, who were familiar with the system where the visualizations were intended to be integrated to. There were conversations with them before starting the test, to make the participants relaxed and be open-minded about their opinions regarding the usability of the visualizations, while also collecting some descriptive data. However, the conversations did not involve discussion regarding the visualization. When the participants were testing the visualizations, they were also encouraged to think out loud. The usability test processes were audio recorded. Two participants were not able to attend the test and they completed the testing on their own. The usability test form can be found in Appendix 5.

4 Results

4.1 Development Process

The project started in September 2016, and it was a part-time project for GameRefinery. It started by learning data visualization and Python Bokeh data visualization library. In November, after some prototypes were done and experience with Bokeh was gained, it was realized that Bokeh did not provide enough functionalities that were required by the project. After that different visualization libraries were being considered and compared, and D3js was chosen to be the visualization tool for the project. D3 learning started from mid-November. Until mid-January 2017, many technical difficulties were encountered due to lack of explanation and weak documentation of D3 library. The project paused for two months, but nonetheless the prototype of top 50 grossing games genre visualization was completed. In mid-March the project resumed and the feature breakdown visualization was being worked on, also the top 50 grossing game genres were being finalized. The usability test was conducted by the end of April. The project was planned to start in September 2016 and to finish by February 2017. In comparison with the timeline of the project, excluding the break, it took one more month than planned. The project was completed by 4th of May.

4.2 Visualization Library Comparison Result

When comparing and choosing visualization libraries, they were evaluated from 5 aspects. Web based meant whether the library could be used on a web browser; interactivity checked if the library provides interactivity to the data visualization; documentation was about the usefulness and clarity of its documentation; user control measured the control a programmer had over the generation of visualization, and how many ‘fine-tunings’ and minor adjustments the programmer could do; ease-to-use assessed the accessibility of a library, i.e. whether the library could be learned and implemented quickly and easily; and finally functionality measured how powerful the library was, in terms of visualization possibilities of the library.

For Web based and interactivity, the choices were either yes or no, for the other aspects they were evaluated on a scale of 1 to 5, with 1 being the weakest and 5 being the strongest. Matplotlib must run on Python environment, instead of a web server. Bokeh and Chart.js both were very easy to use and had very good documentation, with relatively good user control. However, Chart.js could only make 8 different kinds of charts, and though Bokeh was more powerful, it still missed the functionality needed for the project. Thus, D3.js was the only option left to be used for the project. Table 2 presents the comparison result.

Table 2 Comparison of selected visualization libraries

	Matplotlib	Bokeh	Chart.js	D3.js
Web based	×	✓	✓	✓
Interactivity	×	✓	✓	✓
Documentation	5	5	5	2
User control	5	3	3	5
ease-to-use	3	4	4	1
Functionality	4	3	1	5

4.3 Discarded Visualizations

Several visualization prototypes and visualization libraries were cancelled due to different reasons. In fact, without enough research, this project started with Bokeh as the visualization library. A very original idea was to visualize the history category trend of top 100 grossing games on the App Store. It was planned to make a streamgraph visualization of top 100 game genres, but Bokeh did not have a function for making a streamgraph. The plotting function for streamgraph was manually written, and the plot result is shown in Figure 20.

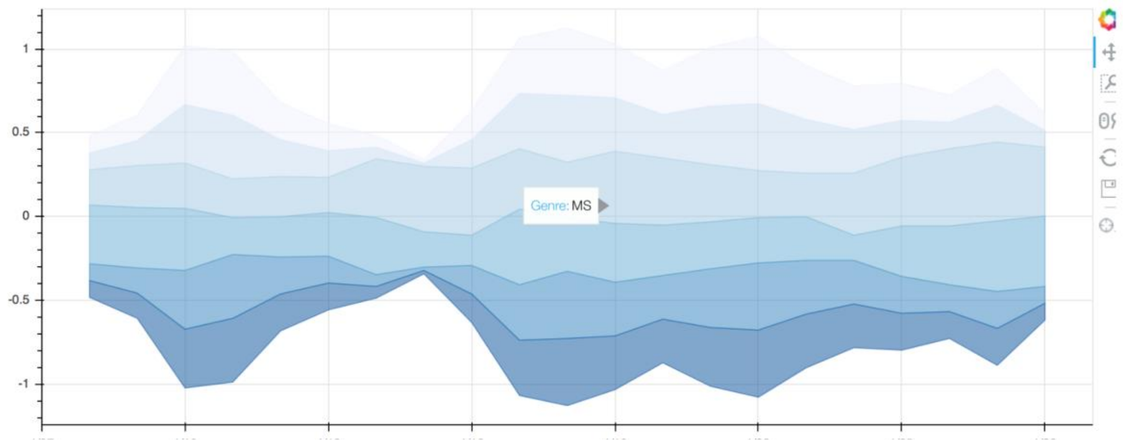


Figure 20 Bokeh streamgraph prototype

As seen in figure 20, the problem with the streamgraph visualization was that when hovering the mouse on top of one point on a specific stripe (which represents the trend of one game genre at a specific time point), it could not display the exact number of games of that genre on the tooltip but only the genre name. There was no solution to this problem as it was answered by one of the developer of Bokeh on a forum that the function used for generating the stripe, `path` function did not support the display of value yet. D3.js did not have a function or layout for streamgraph either, but the functionalities needed is achievable because it provided more controls to the programmer. And at that point, it was realized that the streamgraph visualization of top 100 game genre trends would have a fixed overall “width” with changes only in the partition size of genres and the result would not be as good as expected. This led to a switch from Bokeh to D3.js along with abandoning streamgraph visualization.

After switching to D3, several versions of visualizations were discarded or improved too. Figure 21 shows one version of the top 50 game genres visualization that was replaced by the final version.

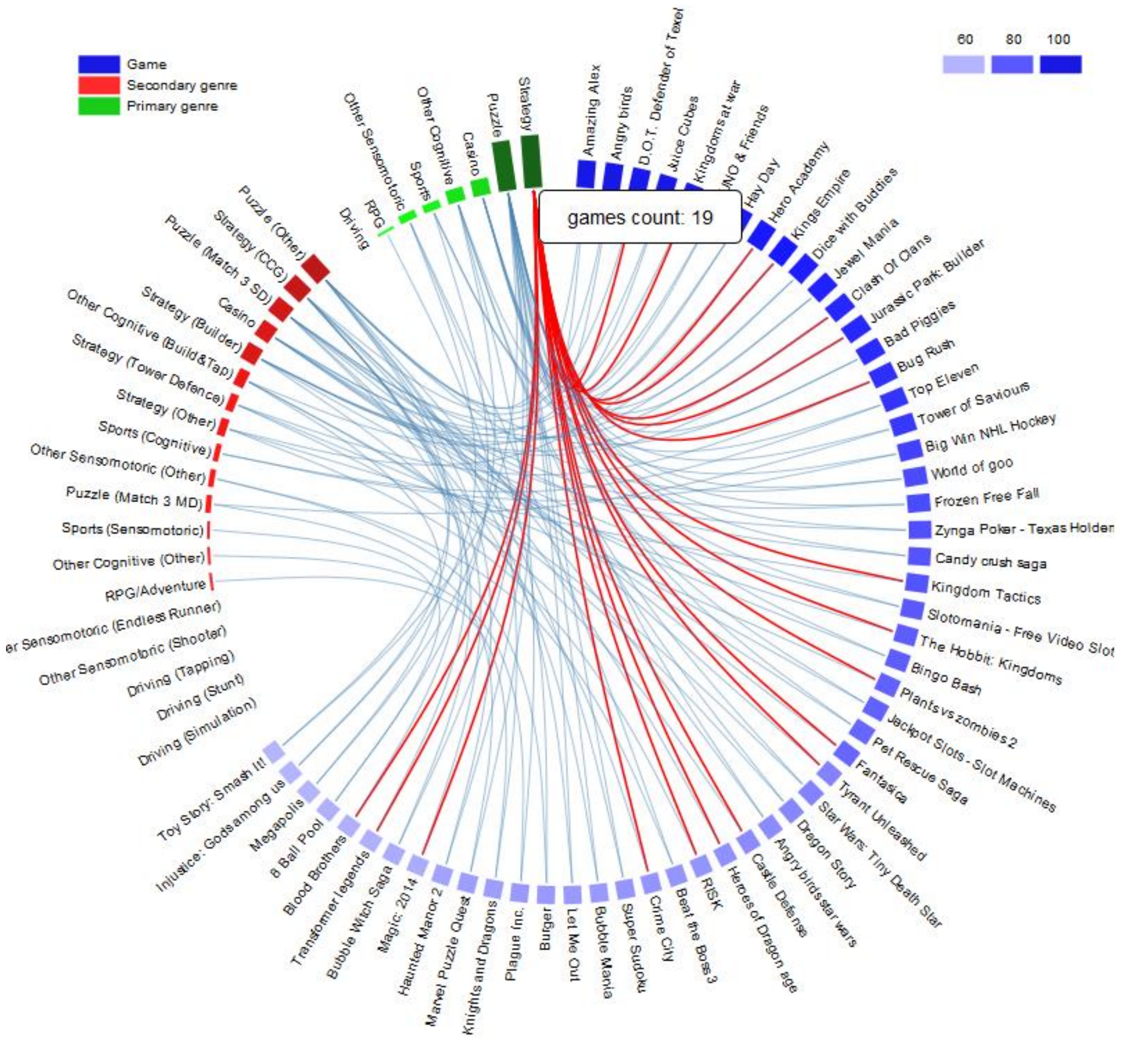


Figure 21. A discarded version of top grossing 50 game genres that provided better clarity

As seen in figure 21, this version, in comparison with the final version, lacked tips on the top left and right. Another difference is that, the links were of the color `steelblue` by default instead of the gradient color mapping. And when hovering the mouse on top of nodes, the links connected to the nodes would be highlighted in red and thickened instead of only thickened in the final version. The decision to change the color was made based on the opinion that the links would present their connection better, which later in usability test, revealed to be otherwise.

4.4 Usability Test Result

The participants of the usability test were analysts from GameRefinery, with 4 of them males and 1 female, and their ages ranging from 23 to 30. Most of the participants had a technical background except for one who had economics background. Prior to their current jobs as analysts now, two participants had game development experience for less than half a year. The “normalized” SUS score ranged from 60 to 92.5, yielding an average SUS score of 81. A SUS score of 80.3 is among the top 10% of scores, based on the results of over 5000 users and 500 different evaluations [25].

The overall results of open questions are affirmative. The feedback for “What were your feelings when you were looking at these visualizations?” were positive, multiple participants stated “good”, “like it”, “wow”.

For the question “What was beneficial about the visualizations in comparison to your previous experience of collecting feature and genre related data?” participants expressed that the visualizations were simple and easy to understand. For the top 50 game genre cluster visualization, the opinion was that it presented a huge amount of data that used to be only perceived from tables, in a very small space. It also helped participants quickly see what genres were trending. Opinion on the feature breakdown treemap was that the color differentiating categories, size and lightness diffracting individual effect of a feature on a game, helped participants observe the importance of a feature category and individual feature fast and easily.

Feedback on question “What was confusing/misleading about these visualizations?” also revealed problems. The feedback on the feature breakdown treemap visualization were that there were too many squares. Some features were too small, and it was hard to hover on top of the small squares to see the pop-up detail, and hard to see the details overall. Feedback on the top 50 game genres cluster visualization were that there were too many links colored similarly, causing it hard to tell which line was leading where. Also, the game names were displayed radially, leading participants to turn their heads to read the game name.

When asked “What suggestions do you have regarding potential improvements to this project?”, participants gave several constructive feedbacks. Two participants suggested displaying more information without hovering or displaying the feature name on the squares treemap feature breakdown. Displaying feature names in the squares was actually one prototype of the visualization but that resulted in the visualization becoming too messy and confusing with too many texts. Another participant suggested displaying the feature name elsewhere outside the visualization, which was considered more reasonable. For top 50 game genres cluster, most opinions were to change the color of links to colder colors.

5 Discussions and Conclusions

The goal of the project was to explore new ways of presenting mobile game analysis data and provide better ways of presenting them to the clients for better game design decision making. The SUS score and user feedback clearly showed that the top 50 game genres visualization helped users pinpoint trending genres among most successful games, and the feature breakdown visualization aided in identifying important individual feature to a game more efficiently and easily, in comparison to how such information was acquired before.

The change of the links color in the top 50 game genres visualization from the discarded `steelblue` mentioned in the Results chapter to the gradient caused participants to feel confused, proved that better graphics does not necessarily provide a better visualization result and help audiences better perceive the data. Now that the project is done, it seems clear that the treemap is probably too colorful and distracting. What potentially could be done for future improvement is a treemap layout that bundles features of the same category together and separates the bundles with some space, so that all the features can have the same color hue to avoid the visualization from being too colorful and distracting.

This project let the author realize his weakness in time management and project design and preparation. If the visualization library comparison had been done properly at the beginning of the project, it would have been done one month earlier. However, this project also allowed the author to learn data visualization systematically and develop new visualization skills.

After research, two ways of visualizing mobile game analysis data, with D3 cluster, bundle and treemap layout were developed, and were proven to be useful. These visualizations can help the clients of GameRefinery make fast and data-driven game design decisions regarding game genre concepts and pinpoint the most important features to focus on when developing or updating a game in the fast-changing mobile game market, where time is of the essence. If similar projects will be carried out in the future, in-depth research on related technologies needs to be conducted during the preparation phase.

Technologies used within the project includes D3.js, Bokeh, and Python. Due to the concept difference D3 presents in comparison to other visualization libraries, it is recommended to use other libraries such as Bokeh, Plotly, Google charts and Chart.js for fast and “traditional” interactive visualizations such as bar chart, line chart and scatterplots. D3.js is recommended for building highly customizable and relatively new visualizations, such as cluster visualization, treemap visualization and Sankey diagram.

6 References

[1] E. McDonald, “The Global Games Market Will Reach $108.9 Billion in 2017 With Mobile Taking 42%,” 20 APR 2017. [Online]. Available: https://newzoo.com/insights/articles/the-global-games-market-will-reach-108-9-billion-in-2017-with-mobile-taking-42/.

[2] M. Friendly, “Milestones in the history of thematic cartography, statistical graphics, and data visualization,” 24 August 2009. [Online]. Available: http://www.math.yorku.ca/SCS/Gallery/milestone/milestone.pdf. [Accessed 8 May 2017].

[3] “An Overview of Suzhou Astronomical Map Stone Carvings,” [Online]. Available: http://web2.nmns.edu.tw/Web-Title/china/A-1-7_display.htm. [Accessed 8 May 2017].

[4] J.-M. Bonnet-Bidaud, F. Praderie and S. Whitfield, “The Dunhuang chinese sky: a comprehensive study of the oldest known star atlas,” 2009.

[5] University of Illinois at Urbana-Champaign, “Data Visualization Online Course,” [Online]. Available: https://www.coursera.org/learn/datavisualization. [Accessed 8 May 2017].

[6] “The Mandelbrot Set Explorer,” Boston University, [Online]. Available: http://math.bu.edu/DYSYS/explorer/def.html. [Accessed 8 May 2017].

[7] R. L. Devaney, “Unveiling the Mandelbrot Set,” Plus Magazine, 1 September 2006. [Online]. Available: https://plus.maths.org/content/unveiling-mandelbrot-set. [Accessed 8 May 2017].

[8] B. J. Michael Schwarz, Director, Hunting the Hidden Dimension. [Film]. United States: PBS, 2011.

[9] B. P. A. K. S. T. A. Y. Charl P. Botha, “From individual to population: Challenges in Medical Visualization,” 2012.

[10] A. M. Noll, “Scanned-display computer grapihcs,” Communications of the ACM, vol. 14, no. 3, p. 143, 1971.

[11] Python Software Foundation, “Python,” Python Software Foundation, [Online]. Available: https://www.python.org/about/. [Accessed 8 May 2017].

[12] “pandas: powerful Python data analysis toolkit,” 5 May 2017. [Online]. Available: http://pandas.pydata.org/pandas-docs/stable/. [Accessed 8 May 2017].

[13] D. D. E. F. M. D. a. t. M. d. t. John Hunter, “Matplotlib Introduction,” The Matplotlib development team, 2 May 2017. [Online]. Available: http://matplotlib.org/2.0.1/users/intro.html. [Accessed 8 May 2017].

[14] “Welcome to Bokeh,” Continuum Analytics, 2015. [Online]. Available: http://bokeh.pydata.org/en/latest/. [Accessed 8 May 2017].

[15] M. Bostock, “Data-Driven Documents,” 2015. [Online]. Available: https://d3js.org. [Accessed 8 May 2017].

[16] M. Bostock, “D3-3.x-api-reference Layouts,” 24 June 2016. [Online]. Available: https://github.com/d3/d3-3.x-api-reference/blob/master/Layouts.md. [Accessed 8 May 2017].

[17] Z. Lv, “D3.js beginner - 9 layouts,” Our D3.JS, 13 July 2014. [Online]. Available: http://www.ourd3js.com/wordpress/163/. [Accessed 8 May 2017].

[18] M. Bostock, “d3-3.x-api-reference Cluster Layout,” 19 September 2016. [Online]. Available: https://github.com/d3/d3-3.x-api-reference/blob/master/Cluster-Layout.md. [Accessed 8 May 2017].

[19] M. Bostock, “d3-3.x-api-reference Bundle Layout,” 8 December 2016. [Online]. Available: https://github.com/d3/d3-3.x-api-reference/blob/master/Bundle-Layout.md. [Accessed 8 May 2017].

[20] D. Holten, “Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data,” IEEE Transactions on Visualization and Computer Graphics, vol. 12, no. 5, p. 741, 2016.

[21] B. S. B. Johnson, “Tree-maps: a space-filling approach to the visualization of hierarchical information structures,” in Visualization, 1991. Visualization '91, Proceedings., IEEE Conference on, San Diego, CA, USA, 1991.

[22] C. P. Ben Shneiderman, “Treemaps for space-constrained visualization of hierarchies,” University of Maryland, 26 December 1998. [Online]. Available: http://www.cs.umd.edu/hcil/treemap-history/. [Accessed 8 May 2017].

[23] “Data Structures D3.js Accepts,” DashingD3js.com, [Online]. Available: https://www.dashingd3js.com/data-structures-d3js-accepts. [Accessed 8 May 2017].

[24] J. Brooke, “SUS - A quick and dirty usability scale,” [Online]. Available: http://www.usabilitynet.org/trump/documents/Suschapt.doc. [Accessed 8 May 2017].

[25] J. Sauro, “Measuring Usability with the System Usability Scale (SUS),” MeasuringU, 2 February 2011. [Online]. Available: https://measuringu.com/sus/. [Accessed 8 May 2017].

[26] Y. Z. Dong Xu, “MVP: Macromolecular Visualization and Processing,” University of Michigan, [Online]. Available: http://zhanglab.ccmb.med.umich.edu/MVP/. [Accessed 8 May 2017].

[27] M. Kraus, “PIxels covered by a triangle,” 1 June 2011. [Online]. Available: https://en.wikibooks.org/wiki/Cg_Programming/Rasterization#/media/File:Pixels_covered_by_a_triangle.png. [Accessed 8 May 2017].

[28] Z. Lv, “Cluster Layout,” DecemberCafe, [Online]. Available: http://d3.decembercafe.org/pages/layout/cluster.html. [Accessed 8 May 2017].

[29] C. P. J.-D. Fekete, “Interactive information visualization of a million items,” in IEEE Symposium on Information Visualization, 2002. INFOVIS 2002, Boston, MA, USA, 2002.

[30] Z. Lv, “D3.js Advanced — 4.0 treemap,” 4 April 2015. [Online]. Available: http://www.ourd3js.com/wordpress/1024/. [Accessed 8 May 2017].

Appendix 1: Cluster and Bundle Layout Data formatting with Python Pandas

```
import pandas
import json
import numpy.random as rnd

df = pandas.DataFrame.from_csv('dataSet.csv', 0)

primary = df.loc['Primary Game Type ', :]
secondary = df.loc['Secondary Game Type ', :]

#define nodesDict, for all the nodes included in the plot
nodesDict = {'name':'', 'children':[]}

#dict for calculating games count, only for assisting calcula-
tion, is not exported for use in data visualization
categoryDict = {}

#adding game names
for i in df.columns.values[1:]:
    nodesDict['children'].append({'name': i, 'catego-
ry':'game', 'GPS': rnd.randint(60, 100)})

#adding secondary type first and primary type at the end so
that the primary type end up at the very top
(r, c) = secondary.shape
for i in range(r):
    if secondary.iloc[i, 0][0] == ' ':
        if(secondary.iloc[i, 0] == ' Casino'):
            nodesDict['children'].append({'name': second-
ary.iloc[i, 0], 'category': 'secondary'})
            categoryDict[secondary.iloc[i, 0]] = 0
        else:
            nodesDict['children'].append({'name': second-
ary.iloc[i, 0][1:], 'category': 'secondary'})
            categoryDict[secondary.iloc[i, 0][1:]] = 0
    else:
        nodesDict['children'].append({'name': second-
ary.iloc[i, 0], 'category': 'secondary'})
        categoryDict[secondary.iloc[i, 0]] = 0

#primary game type values:
(r, c) = primary.shape
for i in range(r):
    if primary.iloc[i, 0][0] == ' ':
        nodesDict['children'].append({'name': primary.iloc[i,
0][1:], 'category': 'primary'})  #add category name to
nodesDict
        categoryDict[primary.iloc[i, 0][1:]] = 0 #initialize
categoryDict
    else:
        nodesDict['children'].append({'name': primary.iloc[i, 
0], 'category': 'primary'})
        categoryDict[primary.iloc[i, 0]] = 0

#file output
with open('relations.json', 'w') as outfile:
    json.dump(relations, outfile)

#counting how many games fall into each category
for i in range(len(relations)):
    categoryDict[relations[i]['target']] += 1

#adding the calculated sum to the JSON nodesDict
for i in categoryDict:
    for j in nodesDict['children']:
        if j['name'] == i:
            j['gamesCount'] = categoryDict[i]

#file output
with open('nodes.json', 'w') as outfile:
    json.dump(nodesDict, outfile)
```

Appendix 2: Treemap Layout Data Formatting with Python Pandas

```
import pandas
import json

#read in dataset
df = pandas.DataFrame.from_csv('Features.csv', header = 0)

index = []
for i in df.index:
    if i not in index:
        index.append(i)

#formatting data
data = {'name': 'Candy Crush Saga', 'children':[]}
for i in index:
    data['children'].append({'name': i, 'children':[]})

rows, cols = df.shape
index = df.index
for i in range(rows):
    if df.iloc[i, 2] == 1:
        for j in data['children']:
            if j['name'] == index[i]:
                j['children'].append({'name': df.iloc[i,
0],'choice': df.iloc[i, 1], 'value': df.iloc[i, 3]})

#file output
with open('treemapNodes.json', 'w') as outfile:
    json.dump(data, outfile)
```

Appendix 3: Top 50 Game Genres Visualization

```
<!DOCTYPE html>
<html>

<head>
  <style type="text/css">
    .link {
      fill: none;
      stroke-opacity: 0.5;
      stroke-width: 1px;
    }

    .linkHover {
      fill: none;
      stroke-opacity: 1;
      stroke-width: 2px;
    }

    text {
      font-family: sans-serif;
      font-size: 13px;
    }

    .tooltip {
      font-family: sans-serif;
      font-size: 14px;
      max-width: 300px;
      padding: 10px 20px;
      display: block;
      height: auto;
      position: absolute;
      text-align: center;
      border-style: solid;
      border-width: 1px;
      background-color: white;
      border-radius: 5px;
    }
  </style>
  <title></title>
</head>

<body>
  <div id='bundleLayout'></div>

  <script src="https://d3js.org/d3.v3.js"></script>

  <script type="text/javascript">
    d3.json('nodes.json', function(games) {
      d3.json('relations.json', function(relations) {

        games.children.sort(function(a, b) {
          if (a.category == b.category && a.category ==
'game') {
            return b.GPS - a.GPS;
          } else if (a.category == b.category && a.category ==
'primary') {
            return a.gamesCount - b.gamesCount;
          } else if (a.category == b.category && a.category ==
'secondary') {
            return a.gamesCount - b.gamesCount;
          }
        });

        var width = 800;
        var height = 800;
        var r = width / 2 - 200;

        //color gradient
        var blueColor = d3.hsl(240, 1, 0.44); //, 0.8
        var redColor = d3.hsl(0, 1, 0.54); //, 0.8
        var greenColor = d3.hsl(120, 1, 0.39); //, 0.8

        var svg = d3.select('#bundleLayout')
          .append('svg')
          .attr('width', width)
          .attr('height', height);

        //gradiant color map for lines
        var defs = svg.append('defs');

        var primaryGradient = defs.append('linearGradient')
          .attr('id', 'primaryLinearGradient')
          .attr('x1', '100%')
          .attr('y1', '40%')
          .attr('x2', '0%')
          .attr('y2', '60%');

        var stop1 = primaryGradient.append('stop')
          .attr('offset', '0%')
          .style('stop-color', blueColor.toString());

        var stop2 = primaryGradient.append('stop')
          .attr('offset', '100%')
          .style('stop-color', greenColor.toString());

        var secondaryGradient = defs.append('linearGradient')
          .attr('id', 'secondaryLinearGradient')
          .attr('x1', '100%')
          .attr('y1', '100%')
          .attr('x2', '20%')
          .attr('y2', '0%');

        var stop1 = secondaryGradient.append('stop')
          .attr('offset', '0%')
          .style('stop-color', blueColor.toString());

        var stop2 = secondaryGradient.append('stop')
          .attr('offset', '100%')
          .style('stop-color', redColor.toString());

        //data conversion/layout setup
        //because of the size setup, it will be a radial clus-
ter layout, and the x of node will be the degree from positive
x axis, y will be the radius.
        var cluster = d3.layout.cluster()
          .size([360, r])
          .separation(function(a, b) {
            return (a.category == b.category ? 1 : 2) /
a.depth;
          });

        //setting conversion functions
        var nodes = cluster.nodes(games);
        var bundle = d3.layout.bundle();
        var nodeWidth = (r * 2 * Math.PI / nodes.length) - 6;
        var oLinks = map(nodes, relations);
        var links = bundle(oLinks);

        //plotting nodes generated from cluster layout
        function map(nodes, relations) {
          var hash = [];
          var resultLinks = [];

          for (var i = 0; i < nodes.length; i++) {
            hash[nodes[i].name] = nodes[i];
          }
          for (var i = 0; i < relations.length; i++) {
            resultLinks.push({
              source: hash[relations[i].source],
              target: hash[relations[i].target],
            });
          }
          return resultLinks;
        }

        //drawing function
        var line = d3.svg.line.radial()
          .interpolate('bundle')
          .tension(0.85)
          .radius(function(d) {
            return d.y;
          })
          .angle(function(d) {
            return d.x / 180 * Math.PI;
          });

                //label tip nodes
        var colorTipJson = [{
            "x": width / 15,
            "y": height / 15,
            "width": 30,
            "height": nodeWidth,
            "fill": 'hsla(240, 100%, 44%, 0.9)',
            "text": "Game"
          },
          {
            "x": width / 15,
            "y": height / 15 + 15,
            "width": 30,
            "height": nodeWidth,
            "fill": 'hsla(120, 100%, 39%, 0.9)',
            "text": "Primary genre"
          },
          {
            "x": width / 15,
            "y": height / 15 + 30,
            "width": 30,
            "height": nodeWidth,
            "fill": 'hsla(0, 100%, 54%, 0.9)',
            "text": "Secondary genre"
          }
        ];

                  //brightness tip nodes
        var lightnessTipJson = [{
            "GPS": 60,
            "x": width - width / 15 - 70,
            "y": height / 15,
            "width": 30,
            "height": nodeWidth
          },
          {
            "GPS": 80,
            "x": width - width / 15 - 35,
            "y": height / 15,
            "width": 30,
            "height": nodeWidth
          },
          {
            "GPS": 100,
            "x": width - width / 15,
            "y": height / 15,
            "width": 30,
            "height": nodeWidth
          }
        ];

        var gBundle = svg.append('g')
          .attr('transform', 'translate(' + (width / 2) + ', '
+ (height / 2) + ')');
```

```
        //line generation
        var link = gBundle.selectAll('.link')
          .data(links)
          .enter()
          .append('path')
          .attr('class', 'link')
          .attr('d', line)
          .style('stroke', function(d, i) {
            if (d[2].category == 'primary') {
              return 'url(#primaryLinearGradient)';
            } else {
              return 'url(#secondaryLinearGradient)';
            }
          });

                 //nodes appending
        var node = gBundle.selectAll('.node')
          .data(nodes.filter(function(d) {
            return !d.children;
          }))
          .enter()
          .append('g')
          .attr('class', 'node')
          .attr('transform', function(d) {
            return 'rotate(' + (d.x - 90) + ')translate(' +
d.y + ')' + 'rotate(' + (90 - d.x) + ')';
          });

        //append GPS rect for games to nodes
        node.append('rect')
          .attr('class', 'rectangles')
          .attr('width', nodeWidth)
          .attr('height', function(d) {
            if (d.GPS) {
              return d.GPS / 5;
            } else if (d.gamesCount) {
              return d.gamesCount * 2;
            } else {
              return 0;
            }
          })
          .style('fill', function(d) {
            if (d.GPS) {
              return 'hsla(240, 100%, ' + (144 - d.GPS) + '%,
0.9)';
            } else if (d.category == 'primary') {
              return 'hsla(120, 100%, ' + (54 - d.gamesCount *
2) + '%, 0.9)';
            } else {
              return 'hsla(0, 100%, ' + (54 - d.gamesCount *
2) + '%, 0.9)';
            }
          })
          .attr('transform', function(d) {
            return 'rotate(' + (d.x + 180) + ')translate(-' +
nodeWidth / 2 + ', 2)';
          });

        //append text to nodes
        node.append('text')
          .attr('dy', '.2em')
          .attr('transform', function(d) {
            if (d.GPS) {
              return 'rotate(' + (d.x < 180 ? d.x - 90 : d.x +
90) + ')translate(' + (d.x < 180 ? (d.GPS / 5) + 5 : -(d.GPS /
5 + 5)) + ', 2)';
            } else if (d.gamesCount) {
              return 'rotate(' + (d.x < 180 ? d.x - 90 : d.x +
90) + ')translate(-' + (d.gamesCount * 2 + 5) + ', 2)';
            } else {
              return 'rotate(' + (d.x < 180 ? d.x - 90 : d.x +
90) + ')';
            }
          })
          .attr("text-anchor", function(d) {
            return d.x < 180 ? "start" : "end";
          })
          .text(function(d) {
            return d.name;
          });

                  //for testing generated nodes
        nodesT = nodes;
        linksT = links;

                  //appending tooltip
        var tooltip = d3.select('#bundleLayout')
          .append('div')
          .attr('class', 'tooltip')
          .style('opacity', 0.0);

                  //Mouse hover interactivity
        node.on('mouseover', function(d) {
            if (d.GPS) {
              tooltip.html('GPS: ' + d.GPS)
                .style('left', (d3.event.pageX) + 'px')
                .style('top', (d3.event.pageY + 20) + 'px')
                .style('opacity', 1.0);
            } else if (d.gamesCount) {
              tooltip.html('games count: ' + d.gamesCount)
                .style('left', (d3.event.pageX) + 'px')
                .style('top', (d3.event.pageY + 20) + 'px')
                .style('opacity', 1.0);
            };
            link.classed('linkHover', function(l) {
              if (l[0].name == d.name || l[2].name == d.name)
{
                return true;
              }
            });
          })
          .on('mousemove', function(d) {
            tooltip.style('left', (d3.event.pageX) + 'px')
              .style('top', (d3.event.pageY + 20) + 'px')
          })
          .on('mouseout', function(d) {
            tooltip.style('opacity', 0.0);
            link.classed('linkHover', false);
          })

                  //appending explanatory tips
        var colorTip = svg.selectAll('.tipSquare')
          .data(colorTipJson)
          .enter()
          .append('g')
          .attr('class', 'colorTip');

        var tipSquare = colorTip.append('rect')
          .attr("class", "rectangles")
          .attr("x", function(d) {
            return d.x;
          })
          .attr("y", function(d) {
            return d.y;
          })
          .attr("width", function(d) {
            return d.width;
          })
          .attr("height", function(d) {
            return d.height;
          })
          .style('fill', function(d) {
            return d.fill
          });

        var tipNote = colorTip.append('text')
          .attr('dy', '.2em')
          .attr('class', 'text')
          .attr('transform', function(d) {
            return 'translate(' + (d.x + 35) + ', ' + (d.y +
7.5) + ')' //move the text to the right and downward a bit
          })
          .text(function(d) {
            return d.text;
          });

        var lightnessTip = svg.selectAll('.tipLightness')
```

```
          .data(lightnessTipJson)
          .enter()
          .append('g')
          .attr('class', 'tipLightness');

        var lightnessSquare = lightnessTip.append('rect')
          .attr("class", "rectangles")
          .attr("x", function(d) {
            return d.x;
          })
          .attr("y", function(d) {
            return d.y;
          })
          .attr("width", function(d) {
            return d.width;
          })
          .attr("height", function(d) {
            return d.height;
          })
          .style('fill', function(d) {
            return 'hsla(240, 100%, ' + (144 - d.GPS) + '%,
0.9)';
          });

        var lightnessNote = lightnessTip.append('text')
          .attr('dy', '.2em')
          .attr('class', 'text')
          .attr("text-anchor", function(d) {
            return d.GPS < 60 ? "start" : "end";
          })
          .attr('transform', function(d) {
            return 'translate(' + (d.x + 22) + ', ' + (d.y -
10) + ')' //move the text to the right and downward a bit
          })
          .text(function(d) {
            return d.GPS;
          });

      });
    });
  </script>
</body>

</html>
```

Appendix 4: Feature Breakdown Visualization

```
<!DOCTYPE html>
<html>

<head>
  <title></title>
  <style type="text/css">
    .rect {
      stroke: white;
      opacity: 0.8;
    }

    .rectHovered {
      opacity: 1;
    }

    .genreText {
      font-family: sans-serif;
      font-size: 16px;
      fill: black;
      opacity: 0.6;
    }

    .tooltip {
      font-family: sans-serif;
      font-size: 14px;
      max-width: 300px;
      padding: 10px 20px;
      display: block;
      height: auto;
      position: absolute;
      text-align: center;
      border-style: solid;
      border-width: 1px;
      background-color: #F0F8FF;
      color: black;
    }

    .tooltip p {
      color: #1E824C;
    }
  </style>
</head>

<body>
  <div id='treemap'></div>
  <script src="https://d3js.org/d3.v3.min.js"></script>
  <script type="text/javascript">
    var width = 1000,
      height = 500;
    var nodesT;
    //basic setup
    var svg = d3.select('#treemap')
      .append('svg')
      .attr('width', width)
      .attr('height', height);

    //two layers, 1 layer for displaying nodes, layer2 for
displaying genreNode text
    var layer1 = svg.append('g');
    var layer2 = svg.append('g');


    //layout
    var treemap = d3.layout.treemap()
      .size([width, height])
      .padding(1)
      .value(function(d) {
        return d.value;
      });

        //read in json
    d3.json('treemapNodes.json', function(error, root) {
      if (error) throw error;

      var nodes = treemap.nodes(root);

      //leaf node value list
      var valueDict = {};
      var categoryList = [];
      for (i = 0; i < nodes.length; i++) {
        if (nodes[i].depth == 2) {
          if (nodes[i].parent.name in valueDict) {
            valueDict[nodes[i].par-
ent.name].push(nodes[i].value);
          } else {
            valueDict[nodes[i].parent.name] =
[nodes[i].value];
            categoryList.push(nodes[i].parent.name);
          }
        }
      }

          //creating dictionary containing the domain of val-
ues
      var domainDict = {};
      for (i in categoryList) {
        domainDict[categoryList[i]] = [valueDict[cate-
goryList[i]][0], valueDict[categoryList[i]][0]];
        for (j = 1; j < valueDict[categoryList[i]].length;
j++) {
          if (valueDict[categoryList[i]][j] < domainDict[cate-
goryList[i]][0]) {
```

```
            domainDict[categoryList[i]][0] = valueDict[cate-
goryList[i]][j];
          } else if (valueDict[categoryList[i]][j] > domain-
Dict[categoryList[i]][1]) {
            domainDict[categoryList[i]][1] = valueDict[cate-
goryList[i]][j];
          }
        }
      }

          //scaling function definition
      function scaleLightness(domainElement, value) {
        if (domainElement[0] == domainElement[1]) {
          return 45;
        } else {
          return 100 - ((value - domainElement[0]) / (do-
mainElement[1] - domainElement[0]) / 2 + 0.20) * 100;
        }
      }

      //red 0
      //green lawn 90
      //green 120
      //inchworm 84
      //azure 210
      //cyan 180
      //magenta 300
      //orange 65, 30
      //brown 38, 30

      var colors = ['hsla(0, 100%, ', 'hsla(210, 100%, ',
'hsla(120, 100%, ', 'hsla(337, 100%, ', 'hsla(300, 100%, ',
'hsla(180, 100%, ', 'hsla(24, 100%, ', 'hsla(80, 100%, ',
'hsla(60, 100%, '];

          //object containing rectangle colors
      var rectColors = {};
      for (i in categoryList) {
        rectColors[categoryList[i]] = colors[i];
      }

      nodesT = nodes;
      //linksT = links;

          //appending game nodes
      var node = layer1.selectAll('g')
        .data(nodes.filter(function(d) {
          return d.parent && d.value != 0; //filter out first
level node and second  layer nodes
        }))
        .enter()
        .append('g');
```

```
          //appending genre nodes
      var genreNode = layer2.selectAll('g')
        .data(nodes.filter(function(d) {
          return d.parent && d.children; //filter out first
level node and second  layer nodes
        }))
        .enter()
        .append('g');

          //appending game node rectangles
      var rect = node.append('rect')
        .attr('class', 'rect')
        .attr('x', function(d) {
          return d.x;
        })
        .attr('y', function(d) {
          return d.y;
        })
        .attr('width', function(d) {
          return d.dx;
        })
        .attr('height', function(d) {
          return d.dy;
        })
        .style('fill', function(d) {
          if (!d.children) {
            var colStr = rectColors[d.parent.name] + scale-
Lightness(domainDict[d.parent.name], d.value) + '%, 1)';
            return colStr;
          } else {
            return 'white';
          }
        });

          //appending genre node names
      var genreText = genreNode.append('text')
        .attr('class', 'genreText')
        //.style('width', function(d){return d.dx;})
        .attr('text-anchor', 'end')
        .attr('x', function(d) {
          return d.x + d.dx - 5
        })
        .attr('y', function(d) {
          return d.y + d.dy - 5
        })
        .text(function(d) {
          return d.children ? d.name : null;
        });

          //tooltip interactivity
      var tooltip = d3.select('#treemap')
        .append('div')
        .attr('class', 'tooltip')
        .style('opacity', 0.0);

      node.on('mouseover', function(d) {
          if (!d.children) {
            rect.classed('rectHovered', function(l) {
              if (l.name == d.name) {
                return true;
              }
            })
            tooltip.html(d.name + '<br/>' + "<p style=color
:red>" + d.choice + '</p>')
              .style('left', (d3.event.pageX) + 'px')
              .style('top', (d3.event.pageY + 20) + 'px')
              .style('opacity', 1.0);
          }
        })
        .on('mousemove', function(d) {
          tooltip.style('left', (d3.event.pageX) + 'px')
            .style('top', (d3.event.pageY + 20) + 'px')
        })
        .on('mouseout', function(d) {
          tooltip.style('opacity', 0.0)
          rect.classed('rectHovered', false);
        })
    })
  </script>
</body>

</html>
```

Appendix 5: Usability Test Form

top 50 game genres

1. Find out which primary genre is most popular among top 50 games.

2. Find out which secondary genre is most popular among top 50 games.

3. Find out which genres are not popular among top 50 games.

4. Based on the graph, what kind of game genre combination would you think is reasonable and have high revenue potential?

Feature breakdown Treemap

1. What feature genre has the biggest contribution to its revenue potential?

2. What feature genre has the smallest contribution to its revenue potential?

3. At first glance, which feature has the biggest effect on the game potential?

4. If you are developing a game, describe what features you would like to include in your game design.

System Usability Scale

Field of study in university:

Game development experience:

Strongly Disagree — Strongly Agree

1. I think that I would like to use this system frequently	1	2	3	4	5
2. I found the system unnecessarily complex	1	2	3	4	5
3. I thought the system was easy to use	1	2	3	4	5
4. I think that I would need the support of a technical person to be able to use this system	1	2	3	4	5
5. I found the various functions in this system were well integrated	1	2	3	4	5
6. I thought there was too much inconsistency in this system	1	2	3	4	5
7. I would imagine that most people would learn to use this system very quickly	1	2	3	4	5
8. I found the system very cumbersome to use	1	2	3	4	5
9. I felt very confident using the system	1	2	3	4	5
	1	2	3	4	5

10. I needed to learn a lot of things before I could get going with this system