FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Jakub Stacho

Unity UI for real-time plotting of data

Department of Software and Computer Science Education

Supervisor of the master thesis: Dr. Adam Streck
Study programme: Visual Computing and Game Development

Prague 2024

I would like to thank my supervisor, Dr. Adam Streck, for being willing to supervise this thesis, provide me with guidance, and find time to help even with a tight deadline. I would also like to thank my family, who pushed me to finish school and this thesis even in times when I didn't see the light at the end of the tunnel. I would like to dedicate this thesis and my whole effort to my grandfather, as he was the main reason for me to finish this master's degree. I would like to thank my girlfriend Dominika for supporting me through the rough days and nights while working on finishing this thesis. I also cannot forget to thank my friend Mario for his support and great toasts.

Title: Unity UI for real-time plotting of data

Author: Bc. Jakub Stacho

Department: Department of Software and Computer Science Education

Supervisor: Dr. Adam Streck, Department of Software and Computer Science Education

Abstract: Languages used for data science or scientific computing commonly come with a de-facto standard library for plotting, such as GGPlot in R, MatPlotLib in Python, Plotly in Matlab, and others. However, all of these focus on creating singular images from static datasets. In Unity, however, we often need to display data in real-time, at a high refresh rate, and only for the currently relevant subset. Since real-time data typically accumulates by hundreds of data points per second, we must provide the user with appropriate data without losing the app's performance that renders the graphs. This makes rendering graphs in real-time both a UI/UX and an optimization problem. The project aims to develop a Unity package to render basic graphs and charts from real-time data.

Keywords: Unity, real-time plotting, graph charts, optimizations, Unity package

Název práce: Uživatelské rozhraní pro zobrazování dat v reálném čase s použitím Unity

Autor: Bc. Jakub Stacho

Katedra: Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: Dr. Adam Streck, Katedra softwaru a výuky informatiky

Abstrakt: Jazyky používané pro datovou vědu nebo vědecké výpočty běžně přicházejí s de-facto standardní knihovnou pro vykreslování, jako je GGPlot v R, MatPlotLib v Pythonu, Plotly v Matlabu a další. Všechny se však zaměřují na vytváření singulárních obrazů ze statických datových souborů. V Unity však často potřebujeme zobrazovat data v reálném čase, s vysokou obnovovací frekvencí a pouze pro aktuálně relevantní podmnožinu. Vzhledem k tomu, že data v reálném čase se obvykle kumulují o stovky datových bodů za sekundu, musíme uživateli poskytnout relevantné data bez stráty výkonu aplikace, která grafy vykresluje, což z toho činí UI/UX i optimalizační problém. Cílem projektu je vyvinout balíček Unity pro vykreslování základních grafů z dat v reálném čase.

Klíčová slova: Unity, zobrazování grafů v reálnem čase, grafy, optimalizace, Unity balíček

Contents

1 Introduction **8**
1.1 Motivations 8
1.2 Goals 8
1.3 Structure 9

2 Plotting libraries **10**
2.1 Usability 10
2.1.1 Plotting libraries vs. Excel 11
2.2 Static plotting libraries 13
2.2.1 Matplotlib 14
2.2.2 Seaborn 18
2.2.3 Summary 20
2.3 Real-time plotting libraries 21
2.3.1 Interactive plotting libraries 21
2.3.2 Animated plots 24
2.3.3 Comparison with Static plotting libraries 26

3 Unity package **28**
3.1 Games as research tool 28
3.1.1 Unity engine 28
3.2 Package parts 28
3.3 Unity package manager 29
3.4 Importing Unity package 31

4 Requirements and analysis **32**
4.1 Functional requirements 32
4.1.1 Drawing plots 32
4.1.2 Axes 32
4.1.3 Grid 33
4.1.4 Borders 33
4.1.5 Changable data point 33
4.1.6 Aggregation of data 33
4.1.7 Testability 33
4.2 Performance requirements 34
4.2.1 Garbage collector 34
4.2.2 Data storage 34
4.2.3 Rendering 34
4.3 Solution analysis 36
4.3.1 System architecture 36
4.3.2 Rendering 38
4.3.3 Data management 39
4.3.4 Data aggregation 40
4.3.5 Graph components 41

5 Developer documentation 42
5.1 Setting up the project 42
5.2 System architecture 42
5.3 Rendering 43
5.3.1 UIDrawer 43
5.3.2 GraphView 43
5.3.3 LineGraphView 44
5.3.4 BarGraphView 45
5.4 Data handling 45
5.4.1 DataPoint 45
5.4.2 DataCollector 46
5.4.3 CircularBuffer 46
5.4.4 DataViewBuffer 46
5.5 Data aggregation 47
5.5.1 IAggregator 47
5.5.2 BaseDataAggregator 47
5.5.3 Examples 47
5.6 Graph components 48
5.6.1 BordersDrawer 48
5.6.2 GridDrawer 48
5.6.3 GraphLabel 48
5.7 Package unity testing 49

6 User documentation 50
6.1 Importing the package 50
6.2 Sample scene 50
6.3 Assemblying plot 51
6.3.1 Create Canvas 51
6.4 Components 54
6.4.1 LineGraphView 55
6.4.2 BarGraphView 55
6.4.3 DataCollector 56
6.4.4 GridDrawer 56
6.4.5 BordersDrawer 57
6.4.6 GraphLabel 58

7 Evaluation 60
7.1 Performance testing 60
7.2 Limitations 60

8 Conclusion 62
8.1 Future work 62

Bibliography 63

List of Figures 66

List of Tables 67

List of Abbreviations **68**

A Attachments **69**

A.1 Creating Unity package 69

1 Introduction

Nowadays, the world is significant due to the amount of data collected through various methods. A lot of industries are data-driven, and a lot of steps are executed based on collected data. Data are collected all the time and everywhere. When you're crossing a street, you're captured by cameras, and the image is being processed. While you are sleeping and wearing a smart device, data about your sleep, respiration, and other metrics are collected. The most obvious is the case when you are browsing the internet on one of your smart devices. Each of these devices collects a large amount of information about you whenever possible. It was never more important to display data in a form humans can understand in a short time than now. Since 2000, the attention span [1] has decreased from 12 to 8 seconds with the introduction of the new electronic era[2]. If we cannot provide valuable information to the user during this period, the message we want to share might be perceived as unclear or hard to understand.

Programming languages are equipped with robust libraries like GGPlot, MatPlotLib, and Plotly, which are well-known and established for generating static images from static datasets. These tools are essential for creating detailed plots in academic papers, presentations, and reports.

1.1 Motivations

The primary use of plotting libraries is to provide straightforward, logical plots for reports, technical papers, or articles. These plots must be displayed in one view without zooming or panning to help gain information like you often see in the dashboard or interactive plot. However, if we try to display live data in those tools, we will soon realize that it's necessary to make a noticeable effort to make these libraries work. The existing plotting libraries cannot handle such real-time demands as they are not designed for such usage. They are optimized for producing static plots from datasets that do not change once the plotting process begins. This creates problems for applications that display data in real-time, such as those created using the Unity engine. The Unity-developed application requires a dynamic approach as multiple data points can be provided in a short period, so visualizations need to be updated without noticeable delay. Also, the challenge would be to manage system resources effectively to prevent performance bottlenecks.

To address this gap, we need a tool to render basic graphs and charts from real-time data sets that handle the computational load with the need for high-refresh-rate updates, ensuring the visual output remains smooth and responsive.

1.2 Goals

This project aims to develop a Unity package that provides essential real-time tools for plotting graphs with high refresh rates. This package should be helpful to researchers and developers who rely on Unity to write applications that demand

[1]The period during which you can stay interested or listen carefully to something[1].

real-time data visualization without compromising performance. It should also provide a base for eager developers to extend and adjust the package to their uses without significant problems.

1.3 Structure

In Chapter 2, we first introduce plotting libraries and try to explore their uses. We divide them into libraries that provide interactive plots where data stays the same, where the data display is changed to suit the user's needs, and real-time plots, where data changes, and the display of the data changes automatically with them. Then, we introduce packages and explain why we chose Unity while we show how to work with packages in the Unity engine in Chapter 3. In Chapter 4, we analyze the requirements of the final package and possible solutions to the problems that might occur during implementation. In the fifth chapter, we present the reader with the implementation of the package and an explanation of our implementation. Next, we show how to use the package in Unity and describe the parameters of the individual components that can be used to expand plots. We finish this text with a performance evaluation of our package and suggestions for improving it. Finally, we conclude the thesis by summarizing what we achieved and suggesting future work.

2 Plotting libraries

Plotting libraries are tools that can extract essential data from massive data sources and provide only the most relevant information we want to share with the user. This chapter will identify the most popular plotting libraries, examine their use, compare them to well-known programs like *Excel*, and discuss why they became so popular. Try to exploit their shortcomings to evaluate what will be needed and supported in our final project. We will also show how to plot different data in those libraries, provide examples of code used, and discuss possible customization of those plots.

Static and real-time plotting libraries meaning

We will compare static plotting libraries and real-time plotting libraries in this chapter many times, so it's essential to understand what those terms mean.

Static plotting libraries are those libraries that can produce plots where data are static, but the plot is changing to suit the user's needs. Still, those libraries provide little to no interaction with plots or data in real time.

By real-time data libraries, we mean those libraries capable of showing data that changes and the plot is changed with them automatically or that they can provide some user interaction in real time, either by manipulating interactive plots or animating data in the plot.

When we discuss some libraries under the static or real-time libraries section, it doesn't mean they can represent only one of those categories. We chose those libraries and divided them into categories primarily because they are a strong representation of those sections and provide an excellent example to learn what they do, to provide us with inspiration and understanding to implement our plotting package.

2.1 Usability

The vast majority of the plotting libraries allow the display of plots in either 2D or 3D representation. Most of the plots are displayed in 2D. This representation is sufficient to represent most of the data. However, displaying data in 3D might also be a helpful representation as some of the data must be shown in a 3D space because of their nature. For example, those data that require 3D visualization are terrain data, meaning the height in the specific coordinate, as shown in figure 2.1. However, our work will focus more on 2D representation as it is the starting point of most plots.

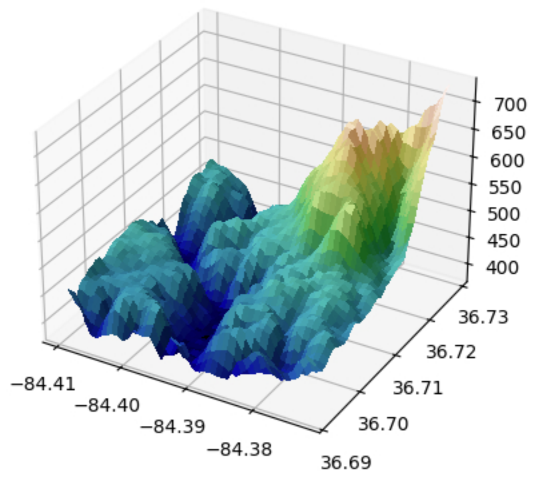


Figure 2.1 Terrain 3D Plot. Source: Matplotlib.org [3]

These plotting libraries are often used in the academic or scientific sphere. Still, we cannot forget the private sector of companies, where the creation of good graphs can provide invaluable insight into problems like cash flow or the company's stock price over time. I think that the usability of these libraries is, in fact, unconstrained. Because almost all libraries provide the functionality to adjust plots to user needs, these plots can be used anywhere they are needed and provide useful insight.

2.1.1 Plotting libraries vs. Excel

The problem that might be for some people who want to create a representation of the data in the form of a graph is the steep learning curve of plotting libraries, which are used within various programming languages such as *Python*, *R*, or *JavaScript*. Those users might lean towards more straightforward tools like *Excel*. In comparison with *Excel*, we chose *Python* as a representative because of the user base, familiarity, and growth in popularity in the last years, based on PYPL index [4]. We will compare the plotting libraries used with *Python* programming language to *Excel* based on the following criteria:

Ease of Use

- *Excel* advantage is in the simplicity of use. Users can create data easily or import them into one of the sheets. They can be organized into tabs that provide more clarity when sorting data. After that, creating a graph from data is simple just by clicking the button in UI.

- Learning any programming language might be frightening and hard for novice users, even though *Python* might be one of the easier ones compared to other programming languages such as C++, C#, or Java. It also takes a while to get familiar with manipulating your data via coding rather than clicking it on *Excel's* UI. Python's steeper learning curve makes it slightly less mainstream as a data analysis tool for the casual user [5].

Scalability

- *Excel* can only handle so much data, and the more data and tabs you have in your workbook, the more difficult it becomes to manage and the slower the file will be. This often leads the program to crash and lose any unsaved work. Excel is not meant to be a full data warehouse with many tables and millions of entries. This gets even slower and more crash-prone when you share the file on the cloud with your teammates. If you're working with millions of rows of data in *Excel*, it would be tough to go up and down through all the rows or create formulas across multiple sheets.

- In *Python*, you can save your data as a separate file and write your code as another file that interacts with the data. This provides many advantages in computational speeds and stability as you potentially wrangle millions of rows of data. In *Python*, the number of rows or columns doesn't change much as the user is not interacting with the data through the UI as in *Excel*. You can still check the data and go through it if you want to see some partial points in the data or correct it. When working across multiple data files, *Python* makes it much easier to merge data from different files on specific fields.

Automation

- Use some sort of automation in *Excel* is possible. We can create macros to do some calculations for us. We can also access *Visual Basic* programming language to write custom scripts and functions to automate tasks. But let's say we need to upload a report every 1st of the month from some data set and create different charts for it. We must do so manually in *Excel*, import new data, and create charts and reports. As small data sets and not-so-complex analysis might still be a way to go, we already feel a lack of features in the automation process.

- Using *Python* in correlation with different libraries makes task automation much easier. The mentioned use case with reports above can be easily scheduled in *Python*. Also, we can integrate with virtually any other system, API or application. Using *Jupyter Notebook* package for *Python* can simplify automation tasks as simple as clicking the 'Run' button and running the entire notebook with multiple data sets and charts as an output. Everything we needed to make in *Excel* could be automated in *Python*

Data connection

Connecting to cloud-based data is possible with *Excel*, but it is a bit more difficult as *Excel* is not meant to stream large amounts of data. *Excel* has made strides in this category, making it a better tool. Still, compared to *Python*, it doesn't have the same level of connectivity or integration capabilities. *Excel* was created long before big data and cloud data platforms were invented. It was originally designed as a financial reporting tool, and it cannot handle the amount of data modern companies use to make business decisions [5]. *Excel* did improve in the last years to integrate *Power Query*, which provides nice graphical UI that allows users to perform data transformations through step-by-step actions. It can be used for simple tasks with relatively small data sets. When processing large data sets, it comes short in performance to *Python*.

Even though using *Excel* is much simpler than learning *Python* programming language and then using some of the plotting libraries, the possibility of these libraries is much greater than what *Excel* provides. There are many things that *Excel* can do. It is an excellent tool for fundamental data analysis. But *Python* allows you to do more regarding analysis, performance, or automation. It all comes down to user needs. *Excel* is an entry-level tool for quickly and easily analyzing a data set. In recent years, there is also the possibility to integrate *Python* into *Excel* to perform some operations over data in sheets. However, if a user is willing to learn something new and dive into the waters of entry-level programming, then they might be rewarded with much more possibilities to process and visualize data sets.

2.2 Static plotting libraries

Static plotting libraries are primarily used to display a single static image as the data representation. You can find many of those libraries, as most are open-source and free. It just depends on your programming language preference, as each of those libraries either has its language or, in most cases, are implemented as a package for already existing programming languages such as *Python*, *R*, or *JavaScript*. The following figure 2.2 presents the most popular plotting libraries based on their respective user base as stated on their GitHub pages.

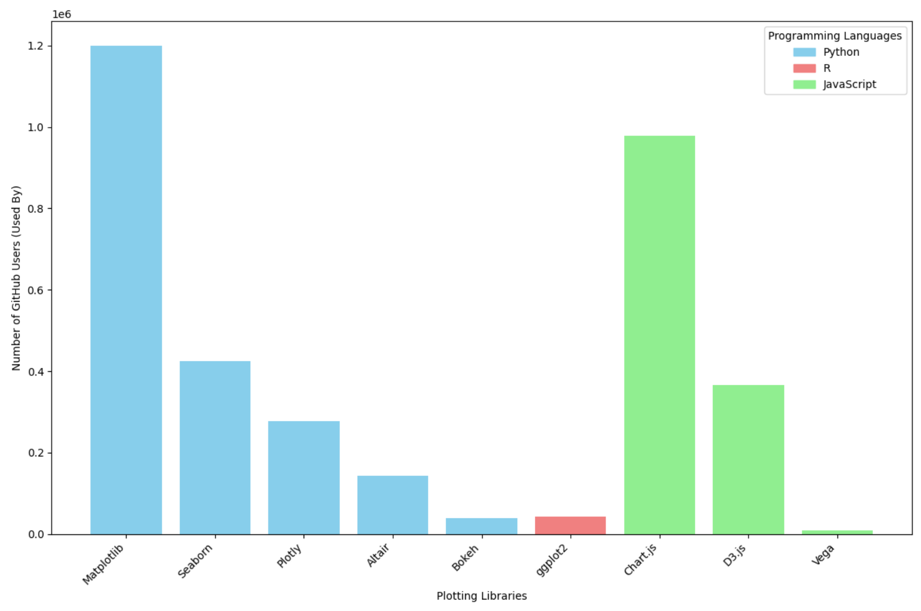


Figure 2.2 Plotting Libraries user base (Obtained from their Github repositories)

Some libraries, such as *Plotly*, might be used with multiple programming languages. In that case, we chose the most used programming language for such a library. As we can see, the most used ones are *Matplotlib* for *Python* and *Chart.js* for *JavaScript.* As we chose *Python* as a representative language in the previous section 2.1.1, we will select two of the most used libraries for plotting in *Python*, which are *Matplotlib*, with over 1 million users, and *Seaborn*, with a little over 400 000.

2.2.1 Matplotlib

Matplotlib is a powerful plotting library in *Python* used for creating static, animated, and interactive visualizations. *Matplotlib's* primary purpose is to provide users with the tools and functionality to represent data graphically, making it easier to analyze and understand. It was originally developed by John D. Hunter in 2003 and is now maintained by a large community of developers [6]. This library has a great number of functions that allow us to create different types of charts or to customize them. For example, the typical representations are line plots, bar plots, scatter plots, histograms, or pie charts. Users can customize basic elements for the chart, like line styles, colors, markers, labels, and many more. *Matplotlib* is designed to produce high-quality plots to be used in academic or scientific reports.

Further, we introduce ways to use this library to generate and customize graphs such as line and bar graphs. We show examples of *Python* code to do so while explaining simple functions. This will be useful as we will see how these plotting libraries are used to be able to design our package in similar ways that users are used to. For the next examples, we will use *PyPlot*, which is the *Matplotlib* module that provides simple functions for adding plot elements, such as lines, images, or text, to the axes in the plot.

Figure

Matplotlib's figure is the top-level container that holds all the elements of a plot. It represents the entire window or page where the plot is drawn. The parts of the basic *Matplotlib* figure are shown in the figure 2.3 below.

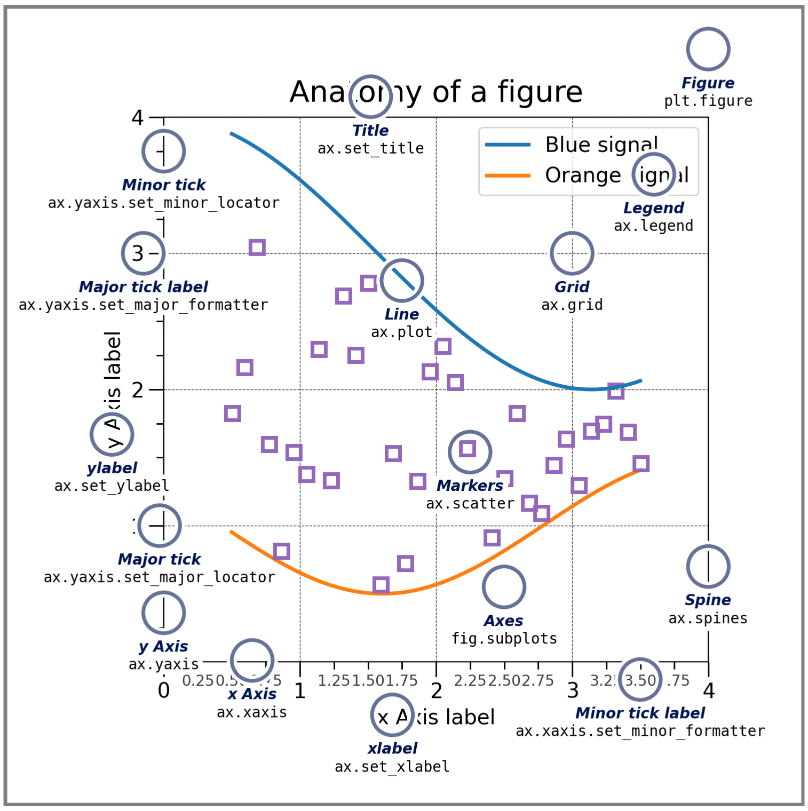


Figure 2.3 Components of a *Matplotlib* Figure. Source: Matplotlib.org [7]

The parts of the *Matplotlib's* figure:

- **Axes** are the rectangular areas within the figure where data is plotted. Each figure can contain one or more axes arranged in rows and columns if necessary. Axes provide the coordinate system and are where most of the plotting occurs.

- **Axis** objects represent the x-axis and y-axis of the plot. They define the data limits, tick locations, tick labels, and axis labels. Each axis has a scale and a locator that determines how the tick marks are spaced.

- **Markers** are symbols used to denote individual data points on a plot. They can be shapes such as circles, squares, triangles, or custom symbols. Markers are often used in scatter plots to visually distinguish between different data points.

- **Lines** connect data points on a plot and are commonly used in line plots, scatter plots with connected points, and other types of plots. They represent the relationship or trend between data points and can be styled with different colors, widths, and styles to convey additional information.

- **Title** is a text element that provides a descriptive title for the plot. It typically appears at the top of the figure and provides context or information about the data being visualized.

- **Axis labels** are text elements that provide descriptions for the x-axis and y-axis. They help identify the data being plotted and provide units or other relevant information.

- **Ticks** marks are small marks along the axis that indicate specific data points or intervals. They help users interpret the scale of the plot and locate specific data values.
- **Tick labels** are text elements that provide labels for the tick marks. They usually display the data values corresponding to each tick mark and can be customized to show specific formatting or units.
- **Legend** provides the key to the symbols or colors used in the plot to represent different data series or categories. They help users interpret the plot and understand the meaning of each element.
- **Grid Lines** are horizontal and vertical lines that extend across the plot, corresponding to specific data intervals or divisions. They provide a visual guide to the data and help users identify patterns or trends.
- **Spines** are the lines that form the borders of the plot area. They separate the plot from the surrounding white space and can be customized to change the appearance of the plot borders.

However, not all of those elements need to be used in each and every graph. *Mathplotlib* gives us the ability to choose which of those elements we want to include or customize. Of course, some of those elements like **Lines** might not give a meaning in the bar plot. In the next examples, we will use the same code base with the same figure setup 1.

Program 1 Base figure setup

```python
import matplotlib.pyplot as plt

# Sample data
categories = ['A', 'B', 'C', 'D', 'E']
values = [10, 24, 36, 40, 29]

# Create a new figure
plt.figure(figsize=(5, 5))
```

Line plot

We will build a line graph representation on the figure set in the program 1. We set *categories* as the x-axis and *values* as the y-axis values, setting the marker type to a small circle. The line would have a classic, uninterrupted style with a green color. After that, on the next line, we set the title to *Line Graph* and labels of the x and y axes. The most important line of code is *plt.show()*, as it is the command to show the final plot.

Program 2 Line graph setup

```
plt.plot(categories,
         values,
         marker='o', linestyle='-', color='green')

plt.title('Line Graph')
plt.xlabel('Category')
plt.ylabel('Values')

plt.show()
```

When we merge program for basic figure setup 1 with line graph representation 2, we get following output: 2.4

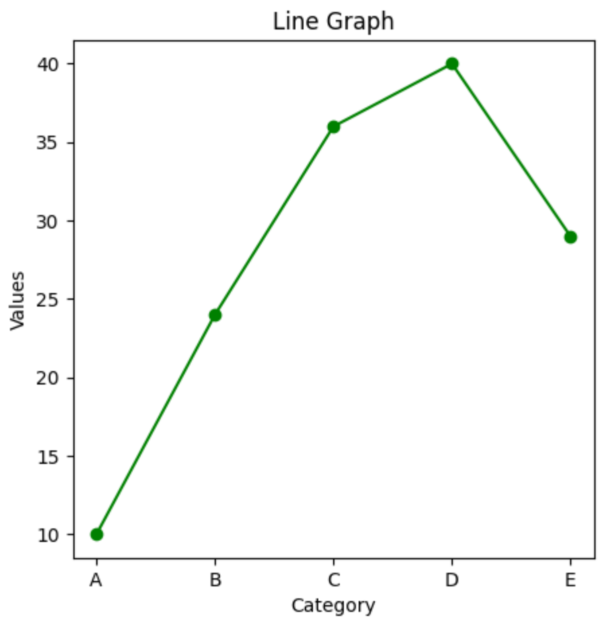


Figure 2.4 Basic line graph using Matplotlib.

Bar plot

As well as the line graph we built in the previous section, we will also build a bar graph representation on the same figure base. We select the x and y axes the same way as for a line graph with defining bar colors.

Program 3 Bar graph setup

```
plt.bar(categories, values, color='skyblue')

plt.title('Bar Graph')
plt.xlabel('Category')
plt.ylabel('Values')

plt.show()
```

When we merge this program with the program setup basic figure 1, we get the following output: 2.5.

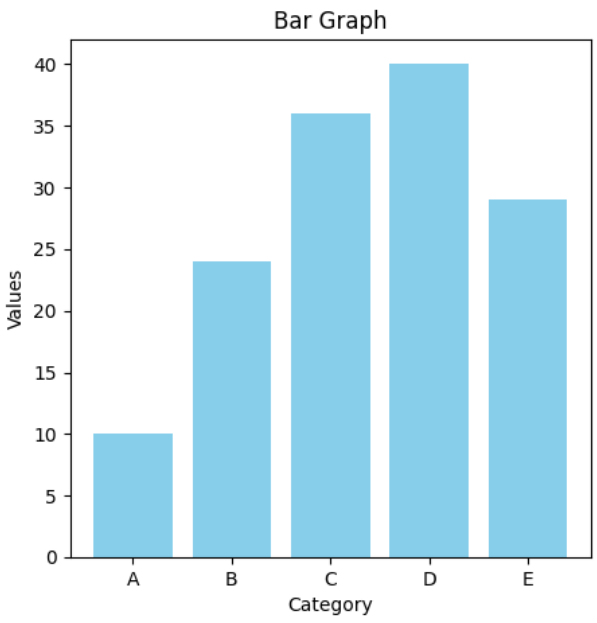


Figure 2.5 Basic bar graph using Matplotlib

2.2.2 Seaborn

Seaborn is a *Python* data visualization library based on *Matplotlib*. When it comes to using *Seborn*, it can act as some kind of wrapper above *Matplotlib*. Creating complicated plots might require a lot of work in *Matplotlib*, while *Seborn* might have some workaround that eases up the workload. *Seaborn's* plotting functions operate on arrays containing whole datasets and internally perform the necessary mapping and aggregation to produce plots. Its dataset-oriented, declarative API lets you focus on what the different elements of your plots mean rather than on the details of how to draw them [8]. As the *Seaborn* is built upon *Matplotlib*, behind the scenes uses *Matplotlib* to draw its plots. In the *Seaborn* user can set the theme of the plot to suit their different need, such as switching between types of plots with different fonts to be able to see them on the presentation, or just in your academic paper. This uses the *Matplotlib rcParam* system and will affect how all *Matplotlib* plots look, even if you don't make them with *Seaborn*. As mentioned above, *Seaborn* is dataset-oriented, meaning it can work with data a little bit more seamlessly as it is designed to work with pandas DataFrame, making it easier to plot data directly from DataFrame. As it might be frightening for novice users to use *Matplotlib* to specify every little thing, *Seaborn* provides an easier setup to create more appealing plots right from the start.

As a base function for plotting graphs in *Seaborn* will be function **relplot()**. This function is named that way because it is designed to visualize many different statistical relationships [8]. In the following code snippet 4 using *Seaborn*, we just provided the names of the variables and their roles in the plot. Behind the scenes, *Seaborn* handled the translation from values in the DataFrame to arguments that

Matplotlib understands. This approach lets the user focus more on what they want to show as a result rather than spending hours writing code.

Program 4 Seaborn replot function call. Source: seaborn.pydata.org [9]

```
# Create a visualization
sns.relplot(
    data=tips,
    x="total_bill",
    y= "tip",
    hue= "day",
    col= "time",
    row= "sex")
```

This simple function call **relplot()** will output the following figure 2.6. As we can see, we just need to specify what represents each column inside of DataFrame data representing the customer's tips in the store. We specified that the x and y axes will be the total amount customers spend and the amount they tip. The hue of the points on the graph will be represented by different days customers left tip. Afterward, we will generate more plots using *col* and *row* parameters to specify how we should represent plots. Each row of the plots will represent a different sex as each column will represent different times customers were in the store.

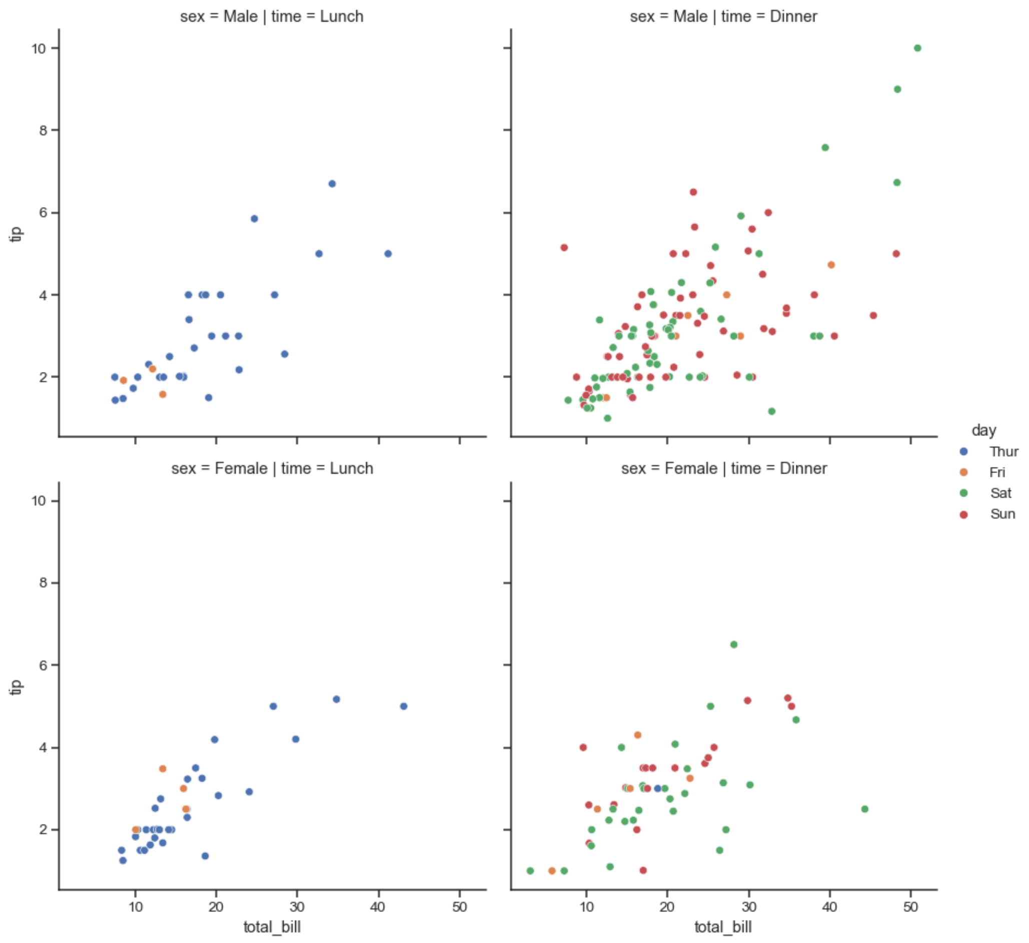


Figure 2.6 Plot representing data of the customer [9].

We showed examples in *Matplotlib* how to make line 2 and bar 3 plots. We can show the same in the *Seaborn*, but it doesn't make sense as it won't showcase *Seaborn's* advantage over *Matplotlib* as the example above 2.6 did. Creation of simple things like line or bar plots are written similarly using either *Matplotlib* or *Seaborn*.

Comparison with Matplotlib

Where *Seaborn* really strives against the *Matplotlib* is just simplicity of the use when trying to visualize datasets into one or more plots. As we were comparing plotting libraries with *Excel* earlier in the chapter 2.1.1, we emphasized the straightforward use of the *Excel* over programming languages using libraries for plotting. I think this advantage is softened up while using *Seaborn* plotting library with *Python*. As we showed earlier in the code snippet 4, we made four plots with a lot of data displayed on them just by calling one simple function.

Creating nicer-looking plots using *Seaborn* right away is also easier than going through a long setup of each element in *Matplotlib*. However, this might be great for casual users. Still, the more experienced and demanding ones will have to write complicated customizations using *Matplotlib* as it is more flexible in this way.

Both of the libraries have their strengths and weaknesses. In my opinion, I would use *Seaborn* when trying to visualize data in nice-looking plots in a short period, with no large need for customization. If I need to customize and fine-tune plots and feel I have more control over the visualization of the data relations, I would use *Matplotlib*.

2.2.3 Summary

Pointing out the advantages or disadvantages of static plotting libraries is possible only from a subjective point of view of what the user needs. When it comes to generating high-quality plots for scientific or academic purposes with no interactivity, static plotting libraries are the way to go. These libraries can be a great fit even in different presentations to create nice-looking plots or while processing large amounts of data to display on one or multiple plots.

This chapter clarifies what we mean by static plotting libraries. We showed a couple of plotting libraries for *Python* and compared them with programs like *Excel* 2.1.1. In the basic examples, we showed the fundamental use of those libraries along with specific code snippets to get particular results. We focused on using those packages, their functions, or the naming conventions users use to work with. We compared *Matplotlib* and *Seaborn* as plotting libraries for *Python*. *Seaborn* is more beginner-friendly and easier to use for users who need to generate nice plots in no time or for those who are not willing to spend their time or energy exploring *Matplotlib* and its functions more deeply. While designing our package, we must balance ease of use with the possibility of plotting data meaningfully or having options to customize plots for more demanding users.

2.3 Real-time plotting libraries

Static plotting libraries provide little to no real-time interactions with plots while visualizing data, as they primarily create unchanging, fixed images. What we mean by real-time plotting libraries is that they are libraries that can provide users with tools to do certain operations in real time. Those operations might be panning, zooming, selecting a specific point in the plot, or viewing data live on the plot as they come to a database. As this project aims to create a package for real-time plotting, we will explore existing real-time plotting libraries and focus on the tools they provide. In this chapter, we will look at those libraries by dividing them into different categories based on their capabilities in terms of real-time plotting.

2.3.1 Interactive plotting libraries

As we explored plotting libraries by popularity and their use in the chapter 2.2, the figure 2.2 also shows plotting libraries *Plotly* and *Bokeh* as quite popular. These libraries can perform certain operations over a plot in real-time, like hover, pan, or zoom. This provides users with much more insight than just reading simple static images. For example, in figure 2.7, we can see that by hovering over a specific point on the plot, we can see exact data on that point. This might be useful if the user needs to read precise data from the plot or make the data presentation more interactive and entertaining.

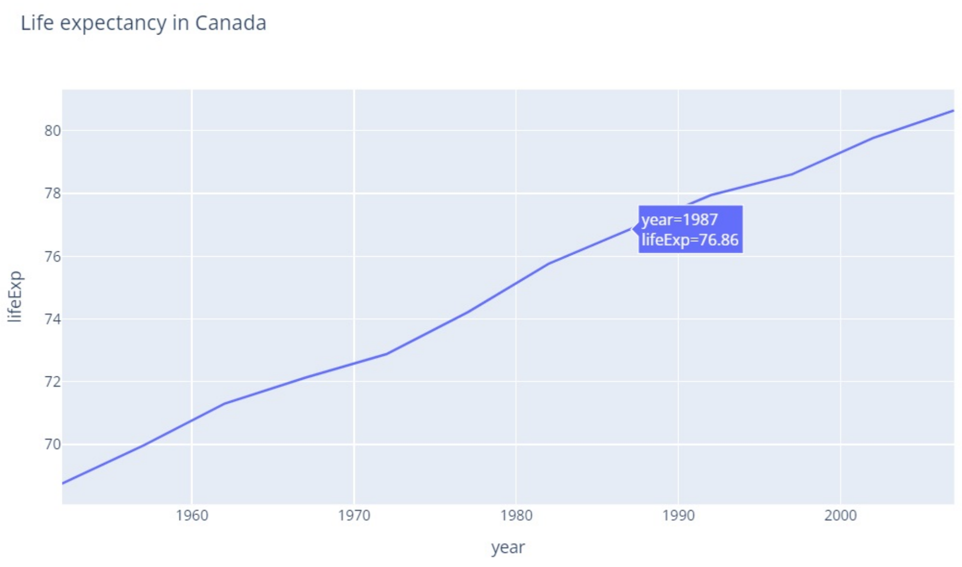


Figure 2.7 Example interactive plot with hover over the specific point. Source: plotly.com [10]

In the figure 2.7, we see an example chart displaying life expectancy in Canada. By hovering over the line on the chart, we can see the label with specific information for that point, precisely the x and y values for the point on the chart.

Plotly

The *Plotly* library in *Python* is an interactive, open-source plotting library that supports over 40 unique chart types covering a comprehensive range of use cases, including 3D charts [11]. We choose a branch of *Plotly* tailored for *Python* users called *plotly.py*.

Plotly is built on top of the *JavaScript* library *plotly.js* to enable *Python* users to create beautiful interactive web-based visualizations that can be displayed in *Jupyter* notebooks, saved to standalone HTML files, or served as part of pure *Python*-built web applications [11].

In the section 2.2, we explored plotting libraries based on their user base. We created a static chart 2.2 using *Matplotlib*. As this chart shows us the most used plotting libraries, we can hardly try to read an exact number of users. In the following figure 2.8, we created the same plot but using *Plotly* library. We can see that when we hover the cursor over the bar representing *Vega* library, it can show us that the number of users is about 9500. This is impossible to read from a static *Matplotlib* chart without providing extra labels in the plot.

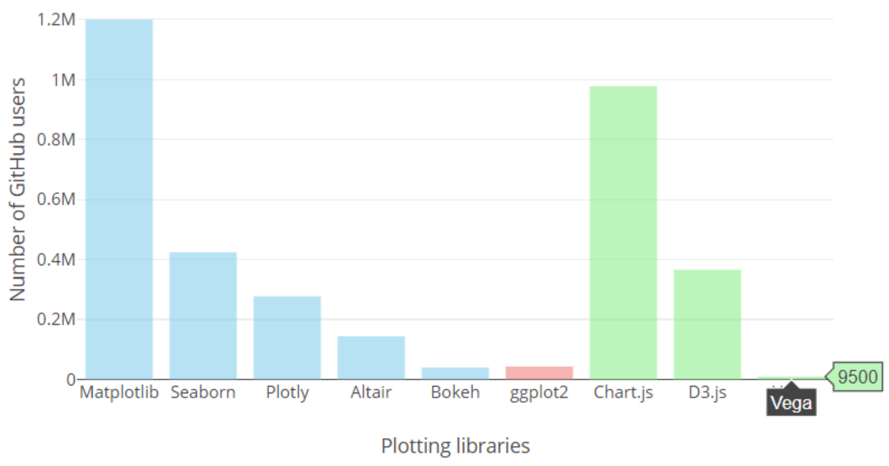


Figure 2.8 Plotting Libraries user base (Created using Plotly)

To generate a simple plot as shown in figure 2.7, *Plotly* required just a little code. It is done similarly to what users are used to in static libraries. In the following program 5, you can see how it can be done in *Plotly*. By writing this couple of lines, *Plotly* will generate the interactive plot for us.

Program 5 Line plot setup in Plotly [10]

```
import plotly.express as px

df = px.data.gapminder().query("country=='Canada'")
fig = px.line(df,
              x="year", y="lifeExp",
              title='Life expectancy in Canada')

fig.show()
```

As we can see, after importing *Plotly's express* package, we load data and provide **line()** function with basic data. We specify what should represent axes and the title or what data to use.

Bokeh

The *Bokeh* is a *Python* library for creating visualizations for web browsers. *Bokeh* will help the user build beautiful graphics from simple to complex plots while managing large data sets with ease and speed [12]. It is an open-source library maintained by different contributors and programmers eager to improve this library. *Bokeh* allows to create interactive *JavaScript* powered visualizations without writing any *JavaScript* code. The idea of *Bokeh* is a process in two steps [13]:

- Select one of the building blocks of *Bokeh* to create visualization.
- Customize these building blocks to fit the needs.

It provides features like *Plotly* in terms of interactiveness with the plot, like zoom, pan, or hovering over specific data points to uncover more detailed information. Based on the user's *Python* code, *Bokeh* performs all the necessary steps when generating *HTML* and *JavaScript* for the user.

Creating interactive plots in the *Bokeh* is also easy and fast. Users require little effort to create interactive plots. In the following code 6, we can see how we can make a simple line graph just by writing a couple of lines.

Program 6 Line plot setup in Bokeh [13]

```
from Bokeh.plotting import figure, show

# Prepare some testing data
x = [1, 2, 3, 4, 5]
y = [6, 7, 2, 4, 5]

# Create a new plot with a title and axis labels
p = figure(title="Simple line example",
           x_axis_label="x", y_axis_label="y")

# Add a line renderer with legend and line thickness
p.line(x, y, legend_label="Temp.", line_width=2)

# Show the results
show(p)
```

This might seem familiar to how we created a line plot in *Matplotlib* by first setting up the figure as a canvas for the plot and then projecting a line on the created figure. The code 6 will result in the following result 2.9. We use simple steps to produce this output [13]:

1. Preparing the data. This can be done by loading some data sets or generating ones using *Python*.
2. Calling the **figure()** function. This call creates a blank canvas (figure) with common default options. To customize the plot, the user can cast the atomized figure's properties, such as title, axes, or labels.

3. Adding renderers. By using **line()** function, *Bokeh* will add the line renderer to the created figure. Renderers have various options to allow users to specify properties like color, legends, or width.

4. Show the results. By calling **show()** or **save()** functions, *Bokeh* will either save the plot into *HTML* or open the file in the browser.

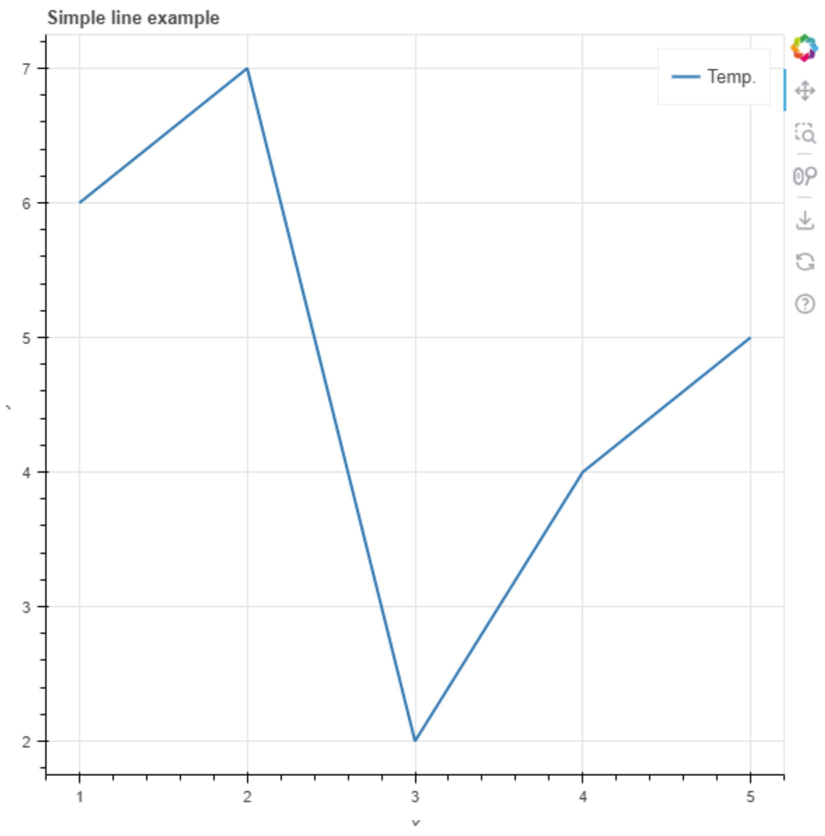


Figure 2.9 Line plot created by using *Bokeh* [13].

2.3.2 Animated plots

Another great possibility that plotting libraries provide is the ability to display animated plots. Animated plots can give insight into how certain data evolved. As human beings, we are naturally more attracted to moving things like points or lines than static images. Animated plots are a great choice to make visualizations more appealing or exciting. Most of the libraries we mentioned so far support animated plots either by default or by importing packages for the library. In those libraries that don't support animated plots, it's still possible to make plots move by wrapping around particular logic to feed data into the plots at different times.

Animated plots in Matplotlib

Even the low-level plotting library *Matplotlib* can be used this way. Based on its plotting functionality, *Matplotlib* also provides an interface to generate animations using the **animation** module. Animations can be done in two ways [14]:

- By using **FuncAnimation** class, we can generate data for the first frame and then modify it for each frame to create an animated plot.
- By using **ArtistAnimation** class, where we generate a list of artists [1], which will be drawn in each animation frame.

Using **FuncAnimation** class is more efficient in terms of speed and memory as it draws an artist once and then modifies it. However, by using **ArtistAnimation** class, we gain more flexibility, allowing any iterable of artists to be animated in a sequence. In the following code snippet, we can see how to create an animated plot using **FuncAnimation** class. We need to provide a figure on which the plot is drawn. Specify the function that will be called every frame. This function would require updating data for different plot elements we want to animate.

Program 7 Animating plot using **FuncAnimation** class in *Matplotlib*

```
# Plot animates for 40 frames with 30 milliseconds delay between
ani = animation.FuncAnimation(fig=fig, func=update,
                              frames=40, interval=30)

plt.show()
```

This might seem pretty straightforward, but we must emphasize that creating a function required by **FuncAnimation** is a nontrivial task for novice users. Creating an animated plot using **ArtistAnimation** is done similarly. Still, instead of a function called every frame, we provide a list of artists that differentiate one from the other.

Animated plots in Plotly

Animated plots can be made more easily with *Plotly*. This doesn't require that much programming knowledge as in *Matplotlib*. On the other hand, it might lack options to customize plots during animations in the way *Matplotlib* provides. It comes down to the balance of library capabilities to ease of use, as discussed multiple times in this chapter. To create an animated plot in *Plotly*, we don't need to change much to the code 5. Several functions of *Plotly* support the creation of animated plots through the **animation_frame** and **animation_group** parameters.

[1]"Almost all objects you interact with on a *Matplotlib* plot are called "Artist" (and are subclasses of the Artist class). Figure and Axes are Artists, and generally contain Axis Artists and Artists that contain data or annotation information." [15]

Program 8 Animating bar plot using **Plotly** [16]

```
import plotly.express as px

df = px.data.gapminder()

fig = px.bar(df, x="continent", y="pop", color="continent",
             animation_frame="year", animation_group="country",
             range_y=[0,4000000000])
fig.show()
```

In the code 8 above, we added **animation_frame** parameter to tell the *Plotly* which parameter should be animated. Also, we provided how the data should be grouped in animation by setting **animation_group** parameter. The disadvantage of this approach is that we always need to set up **range_y** or **range_x** if the plot is also animated on the x-axis. This ensures that the animation will stay in the current figure and can be seen in all frames. The data animation would overflow outside the graph if we adjusted **range_y** in the program 8, for example, to two billion. It may be a problem if we don't know the data beforehand. This code will result in the following plot 2.10.

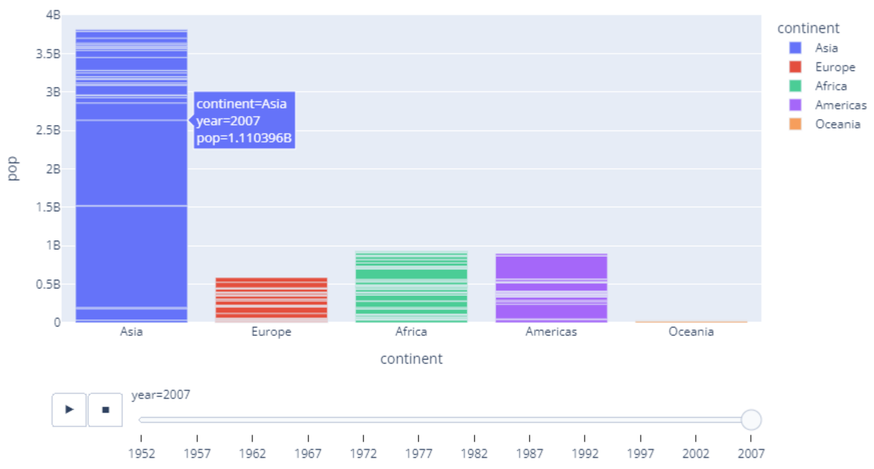


Figure 2.10 Animated bar plot using *Plotly*, generated by code 8. Source: plotly.com [16]

Although *Plotly Express* package supports animation for many chart and map types, smooth inter-frame transitions are only possible for scatter plots and bar plots [16].

2.3.3 Comparison with Static plotting libraries

Comparing libraries, which focus on producing high-quality singular, static images, with libraries that can easily create interactive plots might be challenging as the choice should come down to the user's preference.

Use

When choosing a static plotting library like *Matplotlib*, it can be a good choice for creating static plotting images with a wide variety of customizations. *Matplotlib* offers excellent documentation with a large community and many tutorials as this library becomes standard in the academic and scientific sphere for static plotting images. When users must present data more interactively, real-time plotting libraries like *Bokeh* might be handy, as creating interactive plots is easy and straightforward.

Integration

Integrating plots to web-based applications or showing plots online would be a waste not to use real-time libraries with interactive features like *Plotly* or *Bokeh*. These libraries generate *HTML* and *JavaScript* code so that the user can integrate plots into the web applications. When it comes to exporting interactive plots to standardize more academic document types like *PDF*, there is no option to do so, as *PDF* is not supporting *HTML* code with *JavaScript*. A workaround might be to export plots in different frames and put them into paper, but it would lose the interactability.

Performance

It depends on the size of the data sets the user is working with. *Matplotlib* is more optimized for large data sets when used correctly to generate plots from extensive data fast, as real-time data libraries are not as performant as *Matplotlib*. They were not designed to handle large data sets like *Matplotlib* to render in one static plot. On the other hand, this is mostly not the case for users who need to process massive data sets into interactive plots. We are not claiming that real-time plotting libraries are slow, but compared with *Matplotlib*, creating static plots might not be as efficient.

Shortcoming

Where both groups might come short is real-time live data processing. For animated plots you need to provide data before you create the plot. We are not saying it is impossible to do it, as users would need to develop programs to load live data and project them onto plots, but this is not a task for novice users as most of the libraries would not provide such support. There is room for improvement in this field.

3 Unity package

As we explored plotting libraries in the previous chapter, we may notice that every library comes as a package. Usually, the user needs to download and import a package to use it. This chapter will focus on how to create and import packages in the Unity engine and why we chose the game engine to implement our real-time plotting package.

3.1 Games as research tool

When we first encounter a game engine, we might think that the only purpose of the engine is to create fun games that can provide a pleasant experience. However, using games as a research tool is nothing new. Given the social and economic impact of games, they become essential tools in research. The study of games and games playing is known as *ludology*. "The study of games is the most popular field of research, which engages approaches from anthropology, sociology, psychology, and engineering." [17]. Researchers from multiple fields started promoting game research training and exploring fields, such as gamer creativity, role-play, live-action playing, the concept of the magic circle, and game research methods [18].

We can even find publicly accessible games that a broad audience can play to provide data for researchers. One web page that allows playing games for research purposes is *citizensciencegames.com* [19].

As games are used more for scientific purposes each year, this provides an excellent opportunity to create a package for one of the well-known game engines to provide tools for live data plotting to researchers, who might find this tool helpful when analyzing data.

3.1.1 Unity engine

When choosing game engines, we might consider the two most significant ones used: Unreal Engine and Unity. Choosing one or the other may, in the end, come down to user preferences. The choice is clear, as the author is far more experienced with the Unity game engine when writing this paper.

Unity is a cross-platform game engine developed by Unity Technologies, first announced and released in 2005. The engine has been improved significantly over the years to support various platforms, including desktop, mobile, augmented, and virtual reality. It is considered easy for beginner developers and is famous for indie game development [20]. From my experience over the years of using either Unreal Engine or Unity, the learning curve of the Unity game engine is much less steep than that of the Unreal engine, making Unity an excellent choice for researchers to implement their games for scientific purposes in this engine.

3.2 Package parts

A package usually contains the implementation of the different functionalities and additional data. The Unity package might include other models, textures, animations, etc. The package contains not only important functionality but also additional information, such as:

- **README** file. This file contains information on what the user must do to use the package correctly. It might also contain a link to the web documentation or important notices. Typically, small packages contain only README file, but complete interactive documentation became standard for more extensive, primarily commercial packages.
- **Dependencies** provides information about what other packages are needed for this package to operate correctly. The package typically has dependencies when it uses some other package's functionality, so it becomes dependent on that package. Some of the programs the package is imported with might resolve dependencies independently. This means that the application reads the dependency file, which has different structures for different applications, and imports or updates the packages mentioned in the dependency file.
- **Changelog** is a file where the developer writes changes for each package version. This is standard practice in the field, as users need to know what changes are in the new versions if they want to update it. Sometimes, it might contain breaking changes, which means the user needs to change the implementation when using the package if he decides to update to a newer version.

For a detailed manual on creating a Unity package, see the attachment of this paper A.1.

3.3 Unity package manager

Package Manager is a built-in Unity editor tool. This tool provides management of the packages in the project, as well as the possibility of importing packages or viewing the version history of the package. When we open up the project in Unity, the package manager reads the project manifest (1), a file containing information about what packages are needed in the project. After that, the package manager sends (2) a request to the package registry server (3) for each package in the project's manifest file. The package registry servers then send data back (4), after which the package manager can install or update packages (5) in the project [21]. The following figure 3.1 represents this process of retrieving packages.

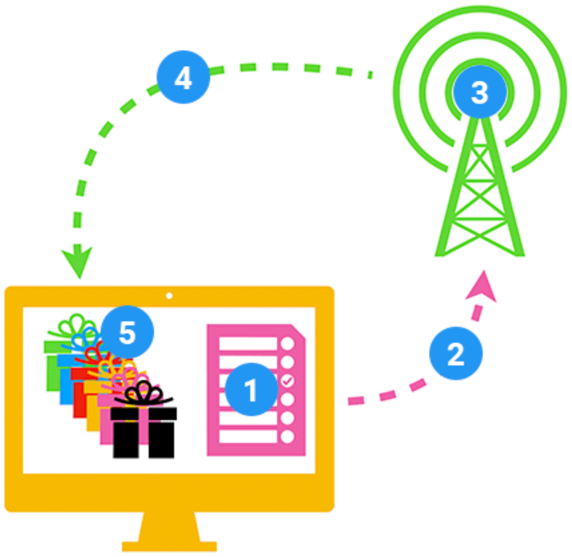
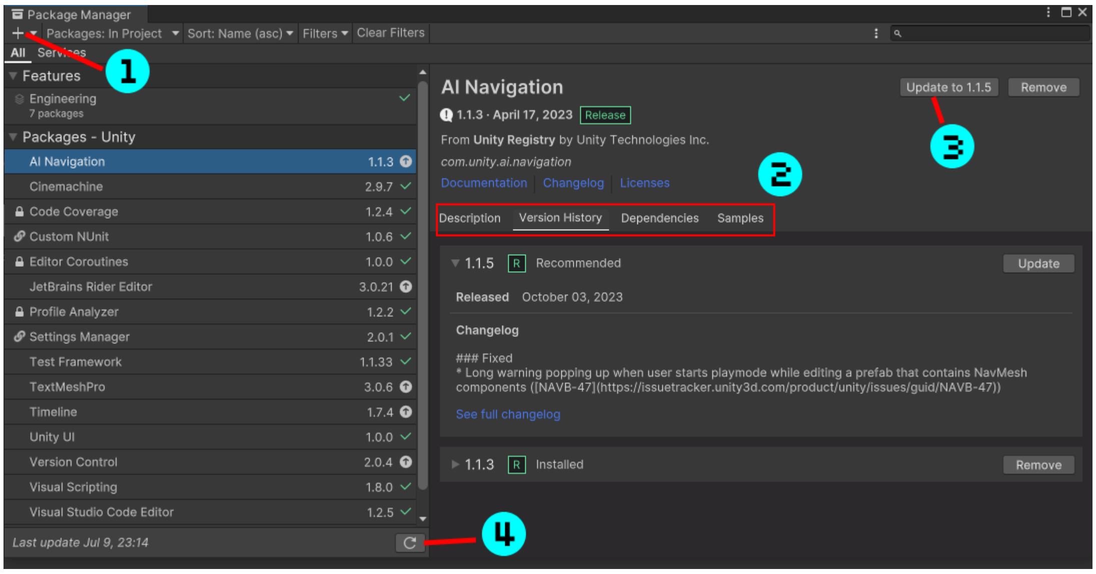


Figure 3.1 Proccess of updating packages in Unity. Source: docs.unity3d.com [21]

We can see the package manager window in the following figure 3.2. On the top left side, there's a button for importing packages (1), either through the URL or to select the package from the disk. When choosing a package in the package list, we can see package detail tabs (2), which display more information about the selected package. We can scroll through these tabs to see information such as description, version history, and dependencies or to import samples provided by the developer. In the top right corner, we can click the button to install or update the package (3). Next to it is a button to remove the package. If we can't see some packages, we can refresh the package list by clicking the reload button at the bottom part of the package manager (4).



Features	
Engineering 7 packages	✓
Packages - Unity	
AI Navigation	1.1.3
Cinemachine	2.9.7 ✓
Code Coverage	1.2.4 ✓
Custom NUnit	1.0.6 ✓
Editor Coroutines	1.0.0 ✓
JetBrains Rider Editor	3.0.21
Profile Analyzer	1.2.2 ✓
Settings Manager	2.0.1 ✓
Test Framework	1.1.33 ✓
TextMeshPro	3.0.6
Timeline	1.7.4
Unity UI	1.0.0 ✓
Version Control	2.0.4
Visual Scripting	1.8.0 ✓
Visual Studio Code Editor	1.2.5 ✓

Last update Jul 9, 23:14

AI Navigation

Update to 1.1.5 Remove

1.1.3 · April 17, 2023 Release

From Unity Registry by Unity Technologies Inc.

com.unity.ai.navigation

Documentation Changelog Licenses

Description Version History Dependencies Samples

1.1.5 R Recommended Update

Released October 03, 2023

Changelog

Fixed
* Long warning popping up when user starts playmode while editing a prefab that contains NavMesh components ([NAVB-47](https://issuetracker.unity3d.com/product/unity/issues/guid/NAVB-47))

See full changelog

1.1.3 R Installed Remove

Figure 3.2 Unity package manager window in the editor.

3.4 Importing Unity package

Unity offers multiple ways to import packages. We will show you several ways to import packages through the Unity Package Manager window. Possible ways on how to import packages are:

1. The easiest way to import packages provided directly by Unity is to click on the dropdown in the left top corner of the package manager window. The dropdown in the figure 3.2 states *Packages: In Project.* We must change this dropdown to *Unity Registry* when importing existing Unity packages. After that, browse the list of available packages and install the desired one.

2. Next way to install the package is to install it from your local disk. When clicking on the plus button (1) in the figure 3.2, choose *"Add package from disk"* from the dropdown. After that, we must select the package's root folder from which we want to import. Then, by double-clicking on the **package.json** file, Package Manager will import the local package and mark it as local. This is the standard way when you download a third-party package that is not officially supported by Unity or is not possible to get on their store page.

3. Another popular way to install a package is to install it from **.git** repository. The process is the same as the previous way, but from the dropdown, we choose to import the package from the git URL. We need to enter a valid git URL pointing to our package's root folder. After that, the Package Manager will import the package and mark it as a gift package. This is a popular way to import packages with an available public git repository, so we don't need to download and import packages from disk every time.

4. The last possible but uncommon way is to import a package by its name. This name is in format *com.company-name.package-name* A.1. After entering the package name, Package Manager will search the package in the Unity registry or the scoped package registry [22]. This is not a used way because knowing the exact package identifier is unusual.

4 Requirements and analysis

As we already explore static and real-time plotting libraries in the chapter 2, we have an excellent overview of current packages' capabilities or what conventions users use. When creating a package in Unity, we need to remember that Unity requires a dynamic approach that can continuously integrate new data points, update visualizations without noticeable delay, and manage system resources effectively to prevent performance bottlenecks. We need to analyze what our package requires in each part of it.

4.1 Functional requirements

We need to define what functions our package will provide to the user and prepare a good base for developers to enhance the package without much struggle.

4.1.1 Drawing plots

The package must be able to render various types of plots to make sure the user can choose how to represent data. The most common ones are line, bar, and scatter plots. Each plot should be customizable with different options for selecting the color of the drawing line, grid, borders, etc.

The solution needs to integrate with Unity's UI system to allow users to render their plots onto UI canvases. To ensure a user-friendly experience, it should be no problem to reposition or scale the UI element the plot is drawn onto.

The package must implement drawing so the user can expand drawing capabilities, such as adding new plot types, without struggle. This ensures that the community can expand the package with ease.

The solution must be capable of updating plots in real-time, which requires redrawing plots frequently. The plots must be drawn in a relatively efficient and performant way to ensure users' seamless experience without any lags.

4.1.2 Axes

The package should display data on the axes with customizable labels, ticks, and scales. The axes should dynamically adjust to handle large datasets and changes in data ranges, ensuring that the visual representation remains accurate and readable. The axes should update in real-time as data changes, maintaining synchronization with the plotted data. This includes dynamically adjusting axis ranges and labels to reflect current data. It would be nice to provide users with customizable axes with various options, like preferred size or scale, with the plot to ensure excellent readability. The user should also have the possibility of having no axes, so the best would be to create the axes as a separate component on which the plot is not dependent.

4.1.3 Grid

Grids provide a reference framework that improves the readability and accuracy of data interpretation in plots. They help users easily connect data points with their respective values on the axes. This feature must be in the package as it provides more clarity for plots.

The grid should be customizable in terms of spacing and color, at least to ensure different user preferences and data visualization needs. Users should be able to turn grid lines on or off and adjust their density for better clarity. Also, the grid should be designed so that it is possible not to include it.

4.1.4 Borders

Borders frame the plot area, providing a clear boundary and improving the visual structure of the plot. They enhance the overall aesthetics and help distinguish the plot from other UI elements that might be present in the application.

Borders should be resizable and customizable, at least in thickness and color. This flexibility allows users to match the plot's appearance with the overall design of their application. This component should be implemented as a separate component that can be connected to the plot.

4.1.5 Changable data point

The package must support real-time updates of data points, allowing users to update the data being visualized dynamically. This is crucial for applications that require monitoring data in real-time. We need to ensure that the solution can handle the large amount of data coming in a short period, and plots must stay responsive and up-to-date with all data updates. This is a crucial feature of our package, as it might differentiate it from others.

4.1.6 Aggregation of data

When handling large data sets, data aggregation is a must to combine multiple data points into single data point representations to simplify visualizations and analysis. Aggregation helps reduce the complexity of data, making it easier to visualize and interpret trends and patterns. What is essential for our real-time package is that it massively reduces the number of data points that need to be visualized.

Our package needs to provide a scalable solution for data aggregation so that users can choose from more types of aggregation. We need to make sure the implementation is written so that any data aggregation solution can be added easily. This flexibility ensures users can tailor the aggregation process to their needs and data characteristics.

4.1.7 Testability

Testable code ensures reliability, maintainability, and ease of debugging. These points are important when developing any software. To provide a good testability of our code, it can reduce our debugging times rapidly. It allows us to verify that each component functions correctly and integrates seamlessly with other system parts.

To ensure that the package is testable, we should maintain principles such as modularity and separation of the components. Each package capability should be in a separate component to ensure we can test its functionality independently. In Unity, we can write editor or runtime unit tests to ensure our package is bug-free.

4.2 Performance requirements

We can't forget about performance requirements. The Unity engine creates applications optimized for real-time interactions, such as games, educational, research, or embedded applications. Performance is crucial. It would be unpleasant if the user found the application laggy after importing our package. The package must be as lightweight as possible, with low memory and rendering requirements.

4.2.1 Garbage collector

One of the problems we might bump into when developing our package in Unity is garbage collection. The garbage collector serves as an automatic memory manager. The garbage collector manages the allocation and release of memory for an application. This means that the developer doesn't need to manage memory himself, like allocating and disposing of the memory the application uses. Automatic memory management can eliminate common problems such as forgetting to free an object and causing a memory leak or attempting to access freed memory for an object that's already been freed [23].

Although it might sound good, when the garbage collector runs and frees up unused memory, it might cause lags in our application because it might take a lot of time to process and free all unused memory. We have to ensure that even if we're allocating some data, we do not assign them to the critical places of the application or allocate too much memory. We can prevent frequent garbage collection by reusing memory or not allocating it too often.

4.2.2 Data storage

The need to create some data storage is obvious. We need to access the data we're receiving or aggregate it to fewer data points from particular data storage. The package should use optimized data structures to handle large volumes of data without performance degradation and frequent need for garbage collection. We need to choose such an implementation of data storage to prevent frequent memory allocation, but at the same time, it must be flexible enough to resize easily if the user needs to.

4.2.3 Rendering

We need to ensure efficient rendering by either exploring existing rendering libraries in Unity for lines and data points, which can help improve performance and reduce development effort, or creating our own rendering of basic lines and points that might be sufficient. Complex or excessive rendering operations, such as drawing too many data points, applying shaders, or performing frequent redraws, can be expensive and negatively impact performance. We need to detect these possible bottlenecks in our application and find workarounds to prevent them from affecting performance. It would be nice to create general ways of rendering lines for developers who want to extend our package by other plot types so they can use it without bothering to fill different buffers or set up vertex indices of triangles.

4.3 Solution analysis

In this section, we must analyze how to approach and implement each part of the package. We go through the architecture and how to approach each package component best and examine it from different angles. In the analysis part, we will try to show more options for implementing the part and explain our choice. Our package implementation is for the Unity engine using C# programming language. The reader should have been familiar with Unity concepts and components to understand this analysis and implementation fully. Even though we try to explain some parts of the chosen solution, we don't dive into the details of basic Unity components.

4.3.1 System architecture

When choosing architecture for our package, we need to think about more possible ways to create plots and how users might use them. We can choose more possible ways how to approach this solution:

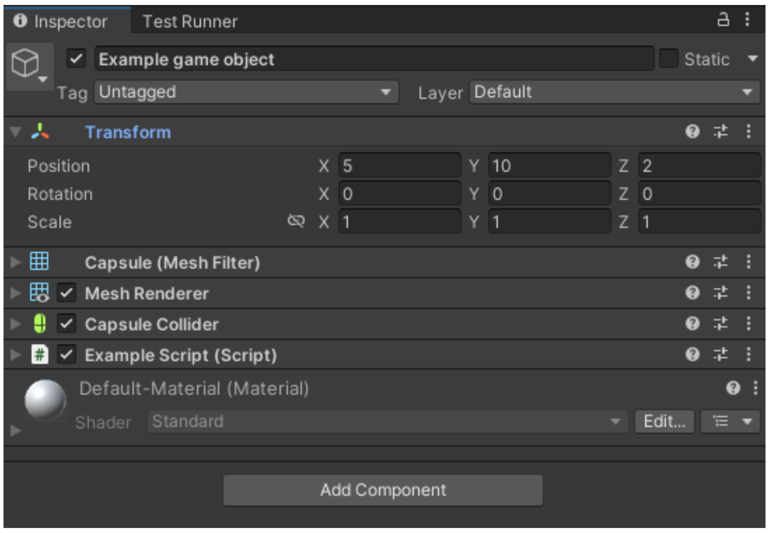
1. More user base approach, where we would create a window that can be manipulated in the way users are used to when manipulating tabs and windows in their operating systems like *Windows*. This window would offer UI for manipulating graph components, such as turning them on or off or customizing each. This window might also be repositioned, closed, or minimalized like the users are used to.

2. Next possibility is to go with one component for each graph type. We would have one script for a line plot and another for a bar plot. This script would also have implemented different graph functionalities and customizations. Users could interact with it through the Unity inspector window to decide what parts of the plot they might want to tailor for their use. We would have to create inspector-friendly components, providing different options for adjusting this script through the inspector.

3. We can use the more harmonious solution with Unity and its components-based systems. Typically, in Unity, the user would assign different components to game objects with various functionalities. This approach allows the user to choose what components to assign to the plot. Each of these components would have its functionality and one purpose. For example, separate axes, borders, or grid components would exist. Also, the user would be able to customize that component as well separately through Unity's inspector window.

All the proposed solutions are different in some ways, and at the same time, they are similar in some ways. The first approach is the most user-friendly, giving us a window with all the functionalities, where the user could click on everything only by using UI. However, this approach has a significant disadvantage in that it probably could not be well integrated with Unity UI and, thus, does not provide other developers with a way to tailor the window for their use. All the functionality would fall under one window, which is a robust solution. Still, the integration problem into Unity UI presents a barrier that the package might be unusable for some applications. The second solution provides the user more flexibility in integrating the package and its components within their application. The user would have to choose the UI component on which they want to draw the graph, and the functionality from our package would take care of that. However, when establishing requirements for this project and sticking to good practice in programming, we would need the solution to be able to be tested by automatic test. This solution wouldn't be easily tested because a lot of functionality would be in one script. The fact that most functionality would be in one script might seem a significant obstacle for users who want to write their solutions by expanding our packaging. However, the third approach combines the benefits of the second approach and some of the benefits of the first. The first approach retains the idea of choosing which components the user wants to put in the graph. However, in this case, it would not be through the window UI but through the unity inspector, where the user could add a component with its functionality that would take care of it. The advantage of the individual component approach is not only the excellent testability of this solution, but it also provides quick debugging because all parts are separate and well debugable. The second approach brings easy integration into Unity UI. This solution is also significant from an extendability point of view because the developer wanting to add their new functions to the package would just be required to write their separate components and add them to the appropriate game object.

For our implementation, we have chosen a third solution, which is not as user-friendly as the first but provides the user with better variability and extensibility for the future. If the user decides to add additional functionality, it would be enough to write another component to be added to the Unity object where the graph is drawn or to reference the Unity UI component where it's drawn. This will also simplify the fact that the user will not have to interfere with the base of this package when expanding or adding new functionalities and providing good testability for each component separately. In the following figure 4.1, we can see the game object in Unity with multiple components like Transform, Mesh Renderer, Collider, or custom script. Each element provides different functionality.


Inspector | Test Runner

✓ Example game object — Static
Tag: Untagged — Layer: Default

Transform
Position X 5 Y 10 Z 2
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Capsule (Mesh Filter)
✓ Mesh Renderer
✓ Capsule Collider
✓ Example Script (Script)

Default-Material (Material)
Shader: Standard — Edit...

Add Component

Figure 4.1 Components assigned onto game object in Unity.

4.3.2 Rendering

One of the most important parts we need to solve is plotting graphs into Unity UI. We need to find a way that is both powerful and easy to use with our package. Also, when expanding the package by other developers, this option should provide as much variability as possible so that they can add their plotting type to the chart without too much trouble. There are several options we can use:

1. Unity already contains a component called Line Renderer [24]. This component takes an array of two or more points in 3D space and creates a line between each one of them. This component already provides much customization, like line width, color, or the ability to assign material to the rendered line.

2. The Unity community has many tools for extending UI in Unity. The popular package publicly available on GitHub called **Unity-UI-Extensions** can be imported into Unity as a package right away. This package offers much more than just a possibility to draw lines. It also provides multiple effects, such as drawing polygons and circles, or comes with many utilities and extensions like bezier paths, sliders, selection boxes, or additional attributes for Unity Inspector.

3. Unity offers a low-level graphics library called **GL**. We can use it to manipulate active transformation matrices or issues rendering commands like in *OpenGL's* immediate mode [25]. This library offers a way to draw lines or other shapes with much customization and variety.

4. One of the possible solutions could be to draw the lines or shapes we need by using the **Graphics** class in Unity [26]. This class serves as a base for all visual UI components. We can create our solution by inheriting the from class and overriding method OnPopulateMesh, where we can provide our raw data regarding vertices and define triangles, which form the mesh to be rendered.

The first option might seem like the easiest one to do, as it comes with Unity. However, this component was created for use in the world space. This doesn't mean it cannot be adjusted to draw in the Unity UI system. Still, when trying to implement this drawing solution, we bumped into a couple of problems, like the possibility of drawing onto canvas in Unity. Also, this solution would be complicated to extend if we implement other types of graphs that can't be drawn using lines. The second solution is using a custom package for Unity, which provides us with many tools, even the ones we don't need at all, like extensions for Unity-specific UI components. When testing this solution for drawing in our implementation, we bumped into the problem of a simple line drawing generating garbage due to ineffective data handling. This package offers what we need and many things we don't need. It seems like too robust a solution to use in our package. The third proposed solution to use **GL** class in Unity can be good for drawing simple things right away but comes with a couple of problems. As Unity states in the documentation [25], it's recommended to draw stuff with **GL** class in OnPostRender stage after the camera renders all required. The problem is that this stage is not supported in the latest render pipelines (HDRP, URP). Also, in the documentation, it is written that this method of drawing is not as efficient as others in Unity, such as rendering custom meshes. The last proposed solution is to use **Graphic** class. This class is tailored for Unity UI components, with the possibility of drawing custom meshes. This method is also used in the second proposed solution for drawing custom shapes and lines. The disadvantage of the drawing with **Graphic** class might be that the method OnPopulateMesh is called by Unity when needed, so we must be careful what operations trigger it not to redraw the whole plot too often.

As a result, we chose the last solution to draw plots with **Graphic** class. This offers us a great way to be in control of drawing different lines or custom shapes. It also provides the best extensibility for the developers or for us to improve the package in the future. It would be just as easy as override method OnPopulateMesh and provide new data in terms of vertices that form a mesh. We need to ensure we're handling data efficiently, not allocating data we don't need or allocating it too often to create a lot of memory to dispose of by the garbage collector.

4.3.3 Data management

We need to think about what we will use data structures for in the first place. We will store raw, live incoming data and want to aggregate them at some point and provide that point to the drawing logic to display it on the plot. We also need to retain a certain number of aggregated points to show some points in the past. This parameter should be variable as the user might want to adjust how many points they want to see on the plot in one frame. We have a couple of options to consider when implementing such data structures:

1. We can use a simple List collection in C#. We would define the data storage size to display the plot in one frame. Also, we could use it to store raw incoming data until they aggregate to a single data point. List offers us the advantage of adding elements dynamically as they come, as well as a couple of functions from C# that can be performed on collections similar to List.

2. Another option is to use a static array with constant size. Whenever we need to add a new element, we copy the last array and its contents into a new one with the new value. The array cannot be resized automatically, so we would need to create an array with a bigger size each time we change how many data points we want to store.

3. We can use some implementation for dequeue abstraction. We can use circular buffer [1]implementation that would be able to handle our needs. We would need to create a fixed-size array in the first place, along with two variables that point to the head and tail of the buffer. We could add data as long as there is free space. This could be handy for data management of viewed points on plots as we need to rewrite the oldest point with the newest and do this in the circle for a fixed data set while the user doesn't want to change the viewed size.

[1]Circular buffer is a data structure that uses a single, fixed-size buffer as if connected end-to-end. This structure lends itself easily to buffering data streams [27].

We can say right away that the second option is not good. We would need to copy arrays all the time, which might be a costly operation when the size of the variety is significant, as this operation has *O(n)* complexity, where n is the size of the array. The first option to use dynamic `List` in C# may seem not so bad in the first place. However, we cannot enlarge the `List` indefinitely as it would potentially take up a lot of memory so that we would need a restriction on the size anyway. The third option is to allocate a constant memory size for the array and two variables for head and tail indices in the buffer. It might have some downsides as the first option if we want to change the number of data points displayed on the plot. In that case, we would need to copy this array into a new one. Also, there might be a problem with a limit for storing raw, live data, so they start to override themselves after the buffer's capacity is full because of the implementation of a circular buffer.

We chose the third option, implementing a circular buffer for our primary data storage. This solution ensures that selecting a reasonable data storage size won't take much more memory. The costly operation of copying the array is only if the user wants to change the number of data points displayed on the plots, which we think won't happen so often. Another mentioned downside was that data are overridden after the buffer is full. As this might seem problematic, users can select the buffer size to suit their needs and balance it before releasing their application.

4.3.4 Data aggregation

We already decided on the data management structure and how we will manage data. We need to think about data aggregation. Data aggregation will need to work with data structures. The question is where we plug the data aggregator in and how often we want to aggregate data. We have multiple options to do so:

1. As we proposed in the system architecture analysis 4.3.1, we can implement a data aggregator as a separate plot component. This would allow the user to choose an aggregation strategy for their needs or implement one just as a separate script. The interval of aggregation would be an exposed parameter for the script.

2. We can plug the aggregator into the script, which will connect data points between our plotting logic and another part of the application. We would need to provide the user with a selection of aggregation strategies in the component inspector window.

Either option might be suitable as there are no significant advantages or disadvantages to either one or the other. The first option seems to fit our system architecture. However, in most cases, the user would want to aggregate data, so making it an optional component might not be the right way. When implementing the second option, we would also need to create some entry points for the data. This might be useful for users when they set up their plots and use just one entry point to feed data. We would need to inject a selection of aggregation strategies in this script to allow users to select some options.

As a result, we chose the second option. Implementing the package as a separate component for each aggregation strategy seems to be too much of the overhead. It might be helpful to consider this option again in the future. We chose the second option to demonstrate the package use and provide the user with a simple entry point to the application. This can be separated in the future to separate aggregation logic from the entry point script. For the initial implementation, we would keep it together for simplicity.

4.3.5 Graph components

As we decided in the system architecture section 4.3.1, we would implement all graph components as a separate Unity component that can be added to game objects. The graph components like grids, borders, or axes would be enough to demonstrate this package. We will show that the other graph components can be implemented similarly and easily integrated into our system. Most of the graph components would need to be drawn onto a graph, so we have a couple of options to do so:

1. We can add each component as a separate drawing part that would inherit from **Graphic** class as we proposed in the section 4.3.2.

2. Component could depend on the drawing component already used for drawing different types of plots. We must create a way to notify all components that provide drawing functionalities like borders or grids.

We can merge both options. Unity allows us to tell the script that is dependent on some other script by using attribute RequireComponent. When we create a drawing logic that will override the OnPopulateMesh method from **Graphic** class as proposed in section 4.3.2, our graph component's script might depend on the logic of this drawing script. The attribute RequireComponent will ensure that the required component is presented on the same game object as our component [1]. This is useful because it provides variability for the user when choosing onto what game object the graph component will be added. Also, it just might be added onto the same game object as, for example, a line drawer, which also requires drawing logic onto the same game object. In this way, the drawing script might be shared.

[1]When you add a script which uses RequireComponent to a GameObject, the required component is automatically added to the GameObject. This is useful to avoid setup errors. When you use RequireComponent, this is done automatically, so you are unlikely to get the setup wrong. [28]

5 Developer documentation

The package was implemented using Unity version 2022.3.2f1. The package is available as an electronic attachment inside **MasterThesis** folder as **realtime-plotting** folder. The structure of the package maintains the recommended structure of the Unity custom packages A.1. In this chapter, we will use different types of font for **game objects**, `source code`, components, or *files*.

5.1 Setting up the project

The recommended way to test the package is to import it into the Unity project using *Package Manager* as proposed in the section 3.3. The package's example is in the **Editor/Scenes/** folder. This is the best way to test the package right away. Open the *GraphScene.unity* and trigger the play mode. You should be able to see the bar and line plot drawings immediately.

5.2 System architecture

As we discussed in the system architecture section 4.3.1, the options we can go through in the implementation and why we chose the component-based architecture. In the following diagram 5.1, we can see the relationships between the different parts of our package. An arrow from one box to another indicates that this part of the package needs to use the functionality of the part to which the arrow is directed to function correctly. We will discuss the different parts in more detail in the other parts of this chapter.

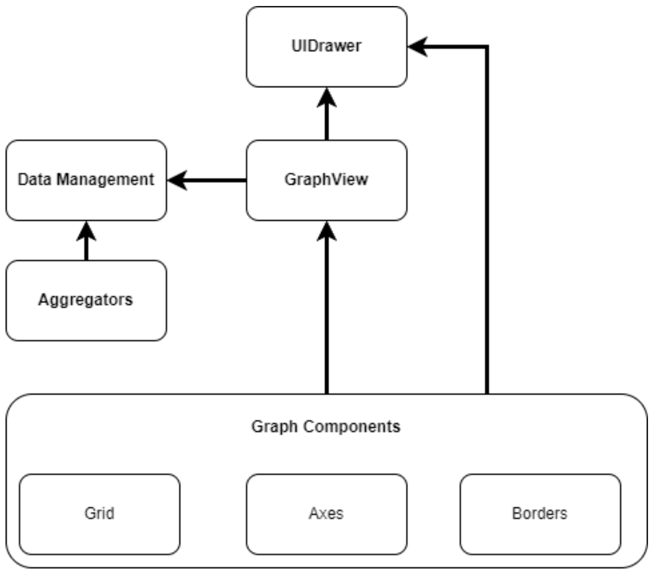


Figure 5.1 Diagram representing relations between parts of our package architecture.

5.3 Rendering

In this section, we will go through the implementation of the rendering of plots. The following scripts can be located in **Runtime/GraphViews** and in **Runtime/GraphDrawers** folders.

5.3.1 UIDrawer

UIDrawer script is a base script that enables us to draw onto the UI. As we proposed in the 4.3.2 section, this script inherits from Graphic class and overrides OnPupalteMesh where we get a reference to VertexHelper class to be able to add mesh data. In the following code 9, we demonstrate the most important parts of this script.

Program 9 Important methods and variables inside UIDrawer script.

```
public Action<VertexHelper> OnDrawGraph;

public void Redraw() => SetAllDirty();

protected override void OnPopulateMesh(VertexHelper vh)
{
    base.OnPopulateMesh(vh);
    vh.Clear();

    OnDrawGraph?.Invoke(vh);
}
```

In the code above 9, we can see defining `Action` delegate with one parameter called `OnDrawGraph`. This is called whenever `OnPopulateMesh` is called by Unity as a process of UI rebuild. This enables us to subscribe to this delegate and draw practically from every script. By calling method `Redraw`, we mark this graphic component of Unity UI as dirty, which forces Unity to call redraw and so `OnPopulateMesh` as a result. In the method `OnPopulateMesh` that we override from Graphic class, we first let the base method perform drawing with provided data. After that, we clear this data and call the delegate to allow other scripts to perform drawing logic.

5.3.2 GraphView

GraphView is an abstract base class designed to be used as a base for different plot types. This class is designed to work also with other components that can enhance the visualization of plots. We chose to do so by implementing an event-based system. GraphView class contains three important delegates:

- `OnInspectorValuesChanged` is called from MonoBehaviour function called `OnValidate`. This is an editor-only function called when the script is loaded or values exposed to the inspector are changed. We added the parameterless delegate `OnInspectorValuesChanged` delegate to allow components to react to changing different parameters of plots. This enables users to debug or fine-tune their real-time plots more easily.

- As other properties can be changed not only by using the inspector window, we created `OnGraphPropertiesChanged` delegate that is called when some crucial properties of the plot are altered to allow other components to react.
- `OnGraphViewSizeChanged` is called in two cases. The first one is at the start of the drawing plot when we don't fill the desired window because we have not yet provided enough data points to draw. The second one is called from function `SetGraphViewSize`, which changes the number of data points the user wants to see on the plot in one frame.
- `OnGraphDataPointsUpdate` is called every time a new data point to draw is added. By subscribing to this delegate, we can react to data change. Useful for components like plot axis.

When using this event, it is essential to note that GraphView class also provides functionality to raise events in the Update loop. This ensures that certain operations are not performed when Unity is in the UI rebuild loop, as it could mess up the rebuilding process. We implemented two methods to ensure this. One is `RaiseEventSafe`, which adds a delegate to a list and ensures this is called just once. The second one is `RaiseStoredEvents` called in the Update loop to ensure the performance of all operations that cannot be done in the UI rebuild loop.

GraphView has a couple of inspector adjustable values. The most important is `viewDataSize`. This indicates how many data points we want to show on the plot. This is an essential value as it influences data store size. This value can also be changed while running in the editor's play mode to adjust the plot more quickly or programmatically to call `SetGraphViewSize`. In either case, changing the `viewDataSize` causes a copy operation of the data buffer, and a new one with a different size needs to be created. This was discussed in the 4.3.3 section. Other adjustable values from the inspector are margins. As the name suggests, we can define an offset from the sides of the UI element RectTransform we are drawing onto. It is also essential for all components to know how to adjust their positioning regarding the plot.

GraphView bsae class is using DataViewBuffer as a data management structure. The base entry points to add data points onto plots are functions `AddDataPoint` for adding a single data point or `ADDataPoints` to add multiple data points providing `IEnumerable` collection.

The essential method that needs to be overridden by child classes is `DrawGraph`. We can implement multiple plot visualizations by overriding this method and providing a suitable implementation for different representations.

5.3.3 LineGraphView

LineGraphView is a simple plot representation that draws lines between data points. This is done by overriding base class method `DrawGraph` and filling `VertexHelper` class with suitable data. On each `DrawGraph` call, we go through all data points to be displayed and draw a line between them. This is done by using `DrawLineSegment` function from RealtimePlottingUtils class. This function creates two triangles for each line, as shown in the following figure 5.2.

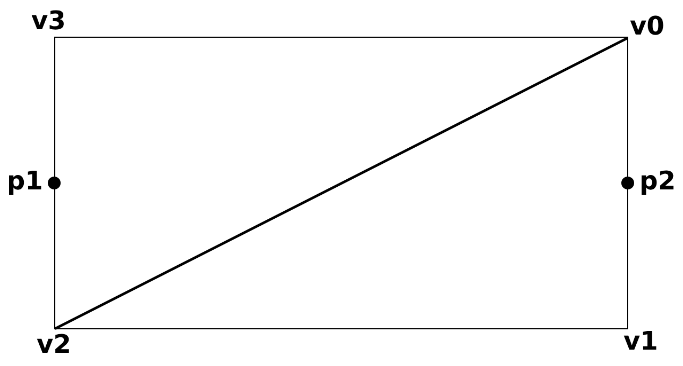


Figure 5.2 Line segment constructed from two triangles and four vertices.

`DrawLineSegment` function gets two points, p1 and p2. Calculates four vertices at a suitable distance to hold the desired width of the line. In the end, two triangles are constructed, consisting of vertices [v0, v2, v1] and [v0, v3, v2]. Note that the sequence of vertices in the triangle is important as the normal vector is determined from this sequence. Constructing triangles in a clockwise direction would cause the triangles not to be visible as the normal vector would point out away from the screen.

LineGraphView also ensures no drawing lines outside the designed area. When a user sets a limiter on the Y axis, some points might be out of bounds of the displaying area; in that case, lines between those points are not shown in the plot.

5.3.4 BarGraphView

BarGraphView works in the similar manner to LineGraphView. The difference is that the bar graph is represented by lines that go from the bottom of the plot to the desired value of the data point on the Y-axis. This script dynamically adjusts spaces between bars based on the number of data points provided. When too many data points are provided, or the desired width of the bar is too wide to fit all bars into the plot, the width is automatically adjusted to preferred displaying all points rather than a few in the desired width.

5.4 Data handling

In this section, we go through the structures used in our implementation as proposed in the section 4.3.3.

5.4.1 DataPoint

DataPoint is a structure that serves us as a representation of the single data point. It implements interface `IComparable` that compares two data points based on Y value. Most of the operation or scaling of the plot is based on the data point Y value. We had to choose one value to be representative while performing operations on data points. Most of the 2D plots contain sorted values on the X-axis and representative values on the Y-axis. This is the reason why we chose this way of implementation. We chose struct implementation over class because we don't want to involve a garbage collector and create data points on the heap. This struct size is 8 bytes because it just contains two floats and no reference types to prevent allocation on the heap.

5.4.2 DataCollector

DataCollector is a component that serves us as an entry point for our plots when we want to use some data aggregation. We can set a couple of inspector values such as `DataFlushInterval` that indicates the time frame after which collected data are aggregated, and a single data point is displayed on the plot. We can also select `AggregationStrategy`. It causes the initialization of the respective data aggregator at the start of the application.

5.4.3 CircularBuffer

CircularBuffer class represents the implementation of the circular buffer. We didn't need to implement it in a dequeue style because adding elements on just one side is enough. It implements `IEnumerable`, allowing us to work with it as with the standard collection in C# like List. CircularBuffer is a generic class with one parameter representing the data type stored in the buffer. This type has to implement `IComparable` interface for us to compare stored values, for example, when trying to find maximal or minimal value. The implementation consists of a fixed-size data buffer and two variables `head` that store the index of the newest element in the buffer and `count` variable that stores how much actual data is stored in the buffer. While implementing this data structure, we tried to keep the standards developers are used to when working with data structures. We provided indexing functionality as a couple of valuable methods like `Peek` that can be used in the same way as for `Queue` data structure in C#.

We can use the `Add` method to add data into the buffer with two overloads. One is for adding a single element at the position of the head in the buffer, and the second is for adding a collection of values. We also provide `Fill` method that fills up an entire buffer with the provided value. When trying to clear the buffer by `Clear` method, all elements will become default values of the data type stored in the buffer.

5.4.4 DataViewBuffer

DataViewBuffer is a special data structure that is used mainly by GraphView. We need to devise an implementation tailored for use in the plot visualization. This data structure allows us to define the size of the viewed data. This site is in the code referenced as **view window**. This data collection stores 2n data points where n is the size of the view window in the `buffer` variable. This eases our implementation. The variable `viewData` is stored in the current view window of data. This was created due to easy manipulation and to provide API to get data currently viewed on the plot. To get those values, we provide method `GetViewData`, which is implemented to perform as few operations as possible. We achieved this by marking our data in the view window as dirty whenever the window needed to be shifted. This ensures that the window values and properties are calculated only once for the same window.

DataViewBuffer also provides min and max values information in the view window. Those are only calculated once the user requests to view window data, not to waste performance when we don't have to. The method for adding values to this collection is `Add` with two overloads for adding a single data point or a collection of data points.

5.5 Data aggregation

To give users and developers some grounding, we implemented simple data aggregators they can work with or inspire when creating their own.

5.5.1 IAggregator

Each aggregator must implement IAggregator interface. This will ensure some API across all aggregators. This interface contains three mandatory functions to implement:

- `Aggregate` function, which is supposed to be used when performing aggreagation. This function should return the last aggregated value.
- `Add` function provides entry points for all aggregators.
- `GetAggreagatedValue` function should implement an aggregation strategy to obtain an aggregated value.

5.5.2 BaseDataAggregator

To provide a base for simple aggregators using some data structure, we created DataAggregator abstract class. This class implements IAggregator interface. It provides a base implementation of the `Add` and `Aggregate` functions, as this might be similar to most of the aggregators. However, we marked these functions as `virtual` to allow developers to adjust the behavior of these functions. The only function we didn't implement in this class is `GetAggregatedValue` as this is specific to each aggregator strategy. This simple base class uses a CircularBuffer for raw data to be aggregated and List data structure to save aggregated values to allow the user to get all aggregated values.

5.5.3 Examples

For example, we implemented two simple aggregation strategies for users to work with some aggregators from the start or for developers to serve as a template when creating their aggregators. We created DataAggregatorSMA to implement a simple aggregation strategy by simply performing average on all raw values and DataAggregatorEMA implementing exponential moving average as an aggregation strategy.

5.6 Graph components

The package has some graph components that help enhance the graphs' customization.

5.6.1 BordersDrawer

BordersDrawer script uses UIDrawer to be able to draw onto the plots by subscribing to `OnDrawGraph` delegate. This component draws four lines to show where the borders are. It does so concerning the set margins of GraphView component.

5.6.2 GridDrawer

GridDrawer script allows users to add a grid to the plot. It works similarly to BordersDrawer and draws a grid by subscribing to `OnDrawGraph` delegate. Users can set up how many lines must be drawn horizontally and vertically. After that, this script calculates offsets between lines and draws them onto the plot.

5.6.3 GraphLabel

GraphLabel is a component representing the axis in the plot. We can choose from the four types of axes based on their placement. To add an axis, we need to place a prefab of the axis as a child of the object where the plot is drawn or into a separate game object. This prefab can be located at **Runtime/Prefabs**, where one is for horizontal axes and the other for vertical. This needs to be done because this functionality requires adding more Unity UI components like LayoutGroup that ensure alignment. We then instantiate simple text labels as the child of this object and will update their contents based on the plot data. The plot axis is resized automatically concerning the data view window size.

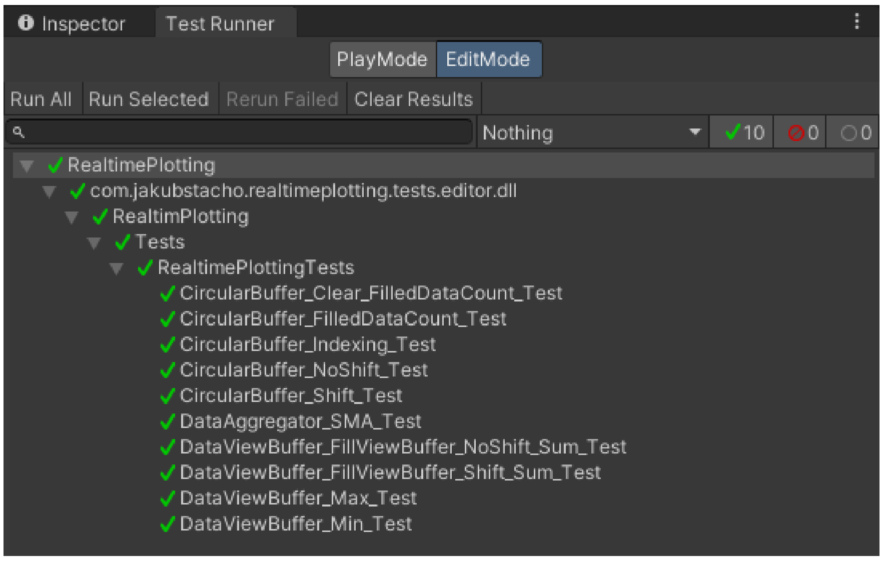
GraphLabel can be customized by defining the preferred size to make sure that labels are seen, or we can set up the preferred count of the labels on the Y axis to ensure values are always seen.

The core functionality of this component is located inside two methods: `UpdateLabelObjectList` and `UpdateLabelValues`. The more frequent one is `UpdateLabelValues`, which is called every time the data point is added to the plot. It adjusts labels on the axis to be aligned correctly and updates the label's contents concerning its position in the plot. `UpdateLabelObjectList` ensures we have just the right amount of labels for displaying values. It is adjusted when we change the number of data points viewed on the graph by creating new labels or turning on already instantiated ones.

Be aware that the current implementation creates a little garbage with each update of labels as we need to convert from decimal type of the data point value to text type using `ToString` method. Strings are immutable in C#, meaning a new one is allocated every time we update a string.

5.7 Package unity testing

This package includes editor tests to ensure the core components' functionality when performing code changes. Tests can be located under **Assets/Tests/Editor** folder in script RealtimePlottingTests. These tests can be run by using Unity Test Runner. In the Unity editor, we can open it under **Window/General/TestRunner** and run edit mode tests. These tests are testing our data collection to ensure their correct functionality, as almost every component of our package uses them. When everything works and is done correctly, the output should look like the following figure 5.3.



```
▼ ✓ RealtimePlotting
  ▼ ✓ com.jakubstacho.realtimeplotting.tests.editor.dll
    ▼ ✓ RealtimPlotting
      ▼ ✓ Tests
        ▼ ✓ RealtimePlottingTests
            ✓ CircularBuffer_Clear_FilledDataCount_Test
            ✓ CircularBuffer_FilledDataCount_Test
            ✓ CircularBuffer_Indexing_Test
            ✓ CircularBuffer_NoShift_Test
            ✓ CircularBuffer_Shift_Test
            ✓ DataAggregator_SMA_Test
            ✓ DataViewBuffer_FillViewBuffer_NoShift_Sum_Test
            ✓ DataViewBuffer_FillViewBuffer_Shift_Sum_Test
            ✓ DataViewBuffer_Max_Test
            ✓ DataViewBuffer_Min_Test
```

Figure 5.3 Unity Test Runner window when every test passes successfully.

6 User documentation

This chapter presents the user documentation for the game. It guides the user through running it and through step-by-step instructions, through which they go through all the essential elements of the package and learn how to use it. When trying to extend the package, see also chapter 5 for a reference and more detailed information about implementation.

6.1 Importing the package

You should import the package into the Unity project using the Unity Package Manager window to use the package correctly. Detailed description on how to do that is in the section 3.4. After successfully importing, the package will be in the **Packages/RealtimePlotting** folder.

6.2 Sample scene

The best place to start exploring the package is to open *GraphScene* inside the package folder in the **Scenes** folder. After opening the scene and running play mode, you should see the window shown in figure 6.1. The bar plot is in the top left corner of the screen, and the line plot is in the bottom right corner.

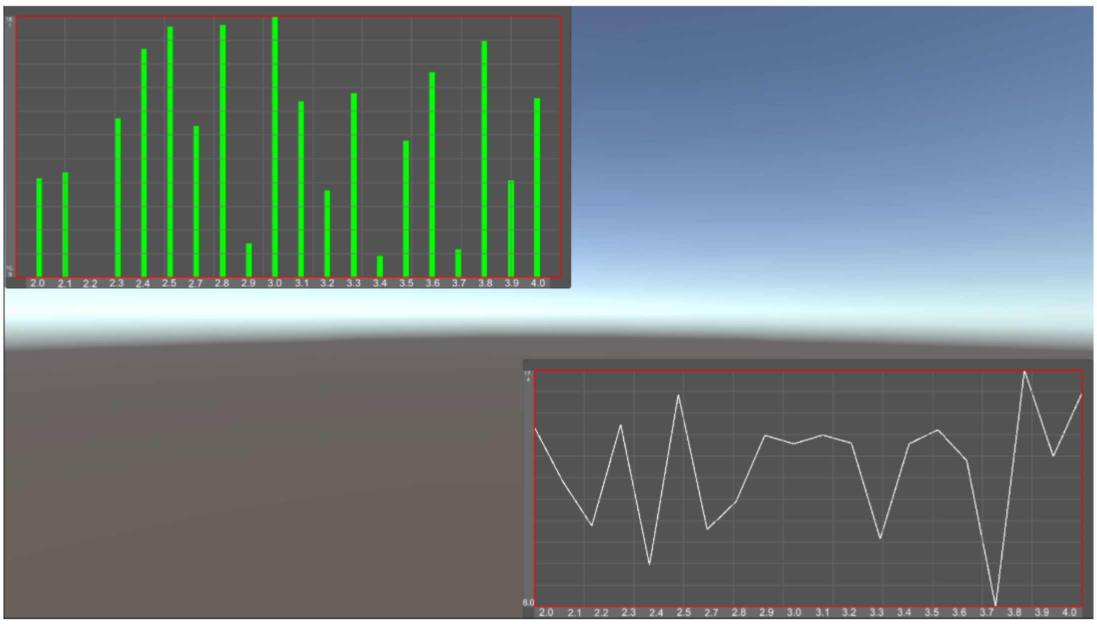


Figure 6.1 Running GraphScene in play mode.

In the hierarchy as present in the following figure 6.2, you can see under MainCanvas game object is located two game objects called BarGraphPanel and LineGraphPanel. Each game object represents an example of how to draw a bar or line plot and assemble it correctly. In the next section, we will dive deeper into how to construct the plot.

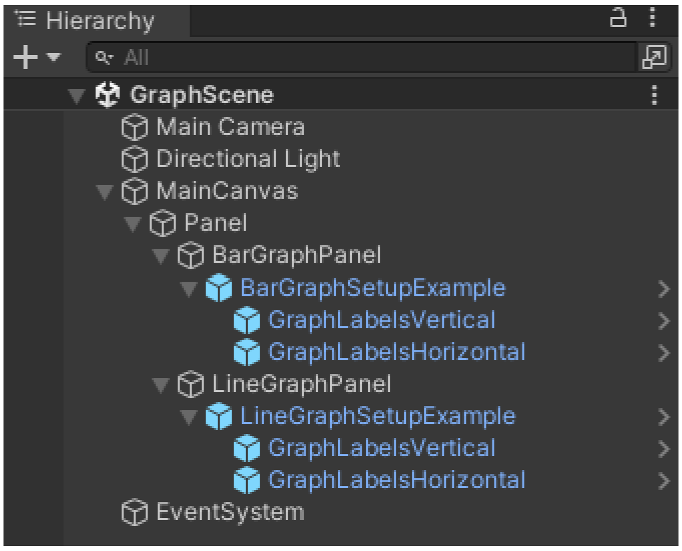


Figure 6.2 Hierarchy of game object in the GraphScene.

6.3 Assemblying plot

In this section, we will go through the process of constructing the plot step by step.

6.3.1 Create Canvas

If you don't have a canvas in the project yet, create one by right-clicking in the hierarchy window and choosing to make a canvas, as shown in the figure below 6.3.

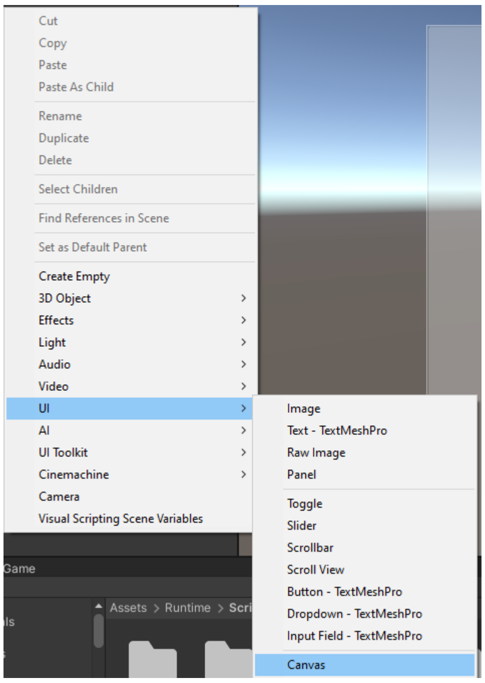


Figure 6.3 Creating canvas in the hierarchy.

Create UI Panel

After creating a canvas in the hierarchy, we must make a UI Panel under the canvas. Please do this by right-clicking on the canvas game object and selecting Panel as shown in figure 6.4. Adjust the panel to your desired size and set its color to the background color of the plot. This panel will serve as the background of the plot.

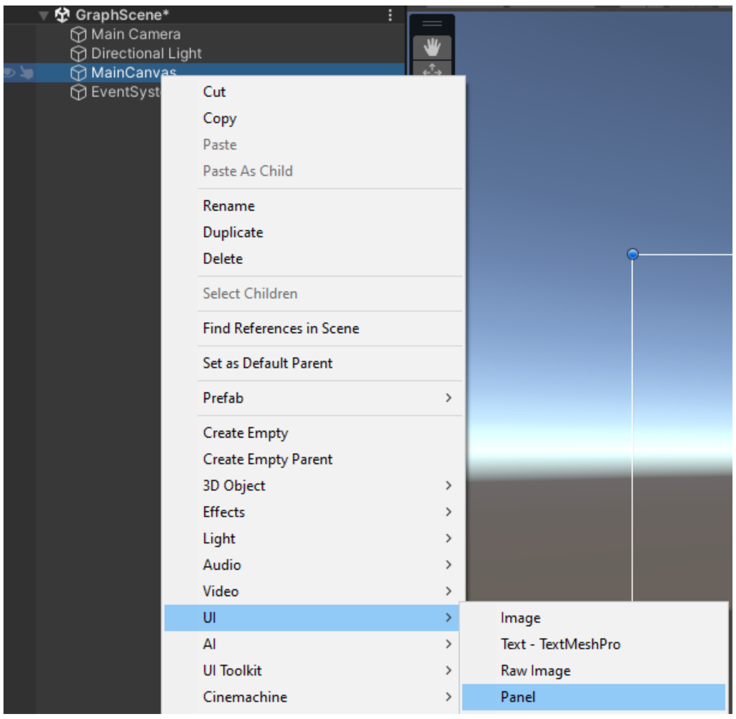
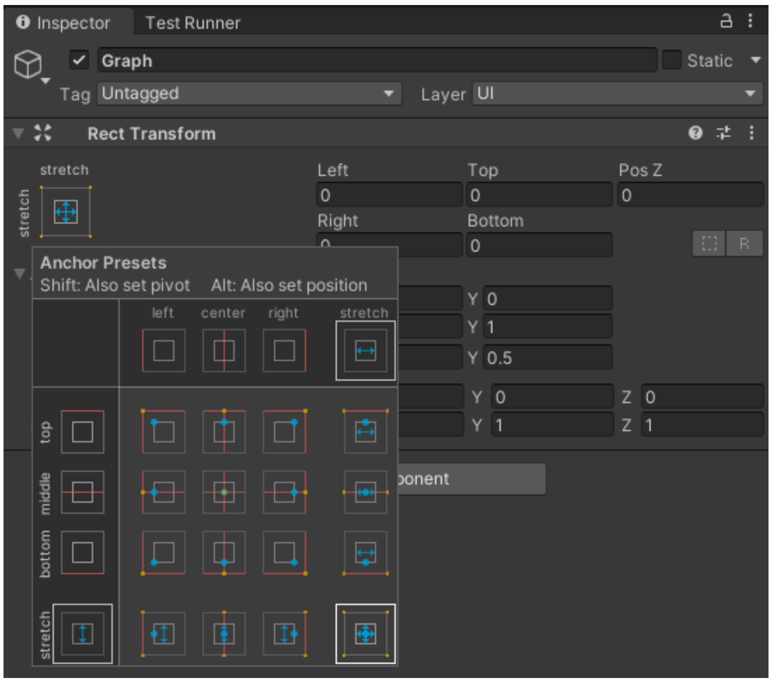


Figure 6.4 Creating panel in the hierarchy under canvas.

Create base graph game object

Under the panel we just created, create an empty game object by right-clicking on the created panel and choosing *Create Empty* from the dropdown. After that, we need to adjust the anchors. Click on the newly created game object and under RectTransform component, and adjust anchors to expand onto their parent, as the figure 6.5 shows. Hold **SHIFT** + **ALT** and select the bottom right button to stretch the RectTransform to the size of the parent panel and set the pivot position to center.

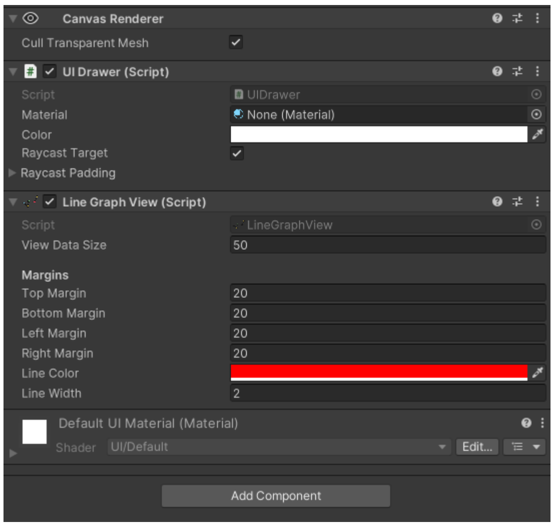


Graph
Tag Untagged
Layer UI
Rect Transform
stretch
Left 0
Top 0
Pos Z 0
Right 0
Bottom 0
Y 0
Y 1
Y 0.5
Y 0 Z 0
Y 1 Z 1

Figure 6.5 Adjusting anchors on RectTransform component.

Adding graph components

We can add various graph components to this game object. At first, we add a component called LineGraphView as shown in the figure 6.6 by clicking on *Add Component* button in the last created game object and selecting LineGraphView.



Canvas Renderer
Cull Transparent Mesh ✓

UI Drawer (Script)

Script	UIDrawer
Material	None (Material)
Color	⬜
Raycast Target	✓
Raycast Padding	

Line Graph View (Script)

Script	LineGraphView
View Data Size	50
Margins	
Top Margin	20
Bottom Margin	20
Left Margin	20
Right Margin	20
Line Color	🟥
Line Width	2

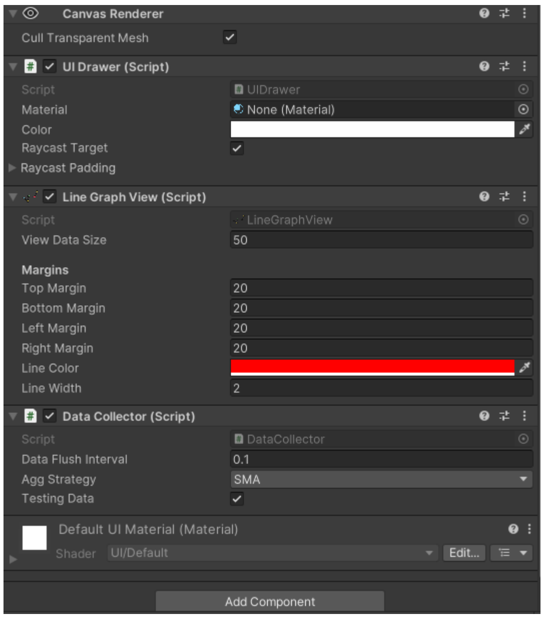
Default UI Material (Material)
Shader UI/Default

Figure 6.6 Components onto game object after adding LineGraphView.

We can notice that UIDrawer component was also added automatically as our graph view component requires this component to be present in the same game object.

Adding graph entry point

However, to be able to see our graph working, we need to provide it with some data. For this purpose, we can use the example DataCollector script. We have to add it onto the same component as our graph view component, as shown in the following figure 6.7.



Canvas Renderer

Cull Transparent Mesh	✓

UI Drawer (Script)

Script	UIDrawer
Material	None (Material)
Color	⬜
Raycast Target	✓
Raycast Padding	

Line Graph View (Script)

Script	LineGraphView
View Data Size	50
Margins	
Top Margin	20
Bottom Margin	20
Left Margin	20
Right Margin	20
Line Color	🟥
Line Width	2

Data Collector (Script)

Script	DataCollector
Data Flush Interval	0.1
Agg Strategy	SMA
Testing Data	✓

Default UI Material (Material)

Shader UI/Default

Figure 6.7 Graph game object after adding DataCollector component.

We can leave default settings and check the box next to `Testing Data` field. This will provide us with random testing data, allowing us to test out graphs quickly. After starting play mode in the editor, you should be able to see the line chart in action as shown in the following figure 6.8.

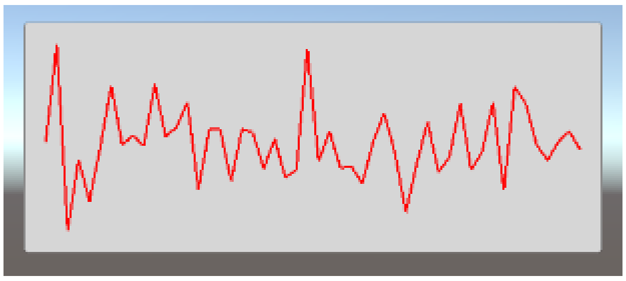


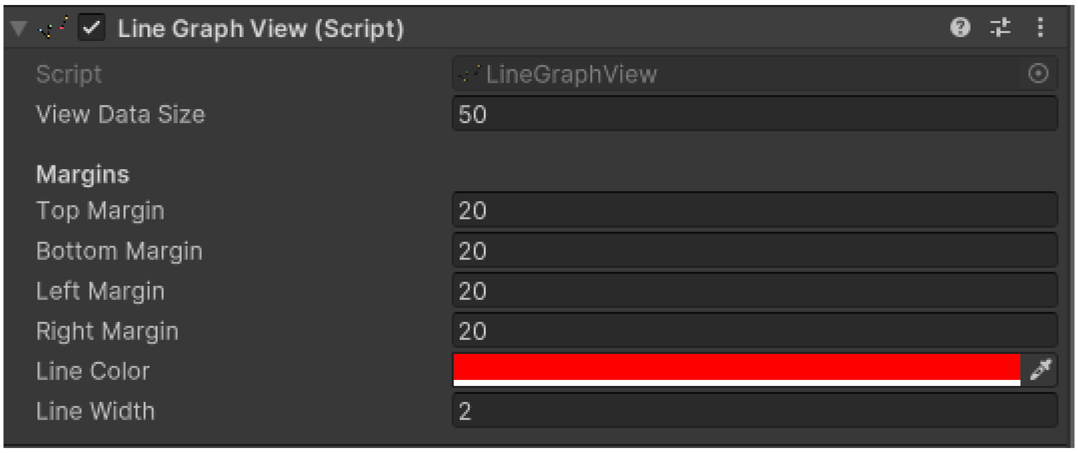
Figure 6.8 Basic line chart in game mode.

6.4 Components

In this section, we will present all graph components in detail and how to use them. We will show examples on the graph created in the previous section 6.3. We also explain how to customize the visual or functional side through the inspector window.

6.4.1 LineGraphView

In the previous section, we saw the LineGraphView component in use. When we add this component, it has a couple of parameters that can be adjusted in the inspector.



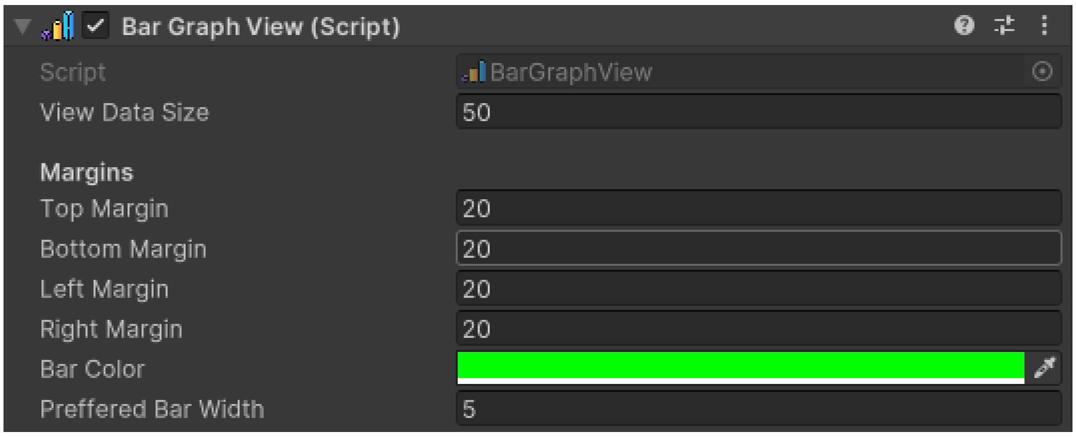
Line Graph View (Script)	
Script	LineGraphView
View Data Size	50
Margins	
Top Margin	20
Bottom Margin	20
Left Margin	20
Right Margin	20
Line Color	🟥
Line Width	2

Figure 6.9 LineGraphView component.

We can adjust View Data Size, which means how many data points we want to display on the plot at the time. We can adjust margins, which causes the graph to be drawn onto a smaller UI panel. Also, we can customize the line's width and color to suit our needs.

6.4.2 BarGraphView

This component works similarly to LineGraphView. The only difference is that we can set the bar width, as we can see in the following figure 6.10.



Bar Graph View (Script)	
Script	BarGraphView
View Data Size	50
Margins	
Top Margin	20
Bottom Margin	20
Left Margin	20
Right Margin	20
Bar Color	🟩
Preffered Bar Width	5

Figure 6.10 BarGraphView component.

When using this variance of GraphView, we can get a similar result as in the figure 6.11.

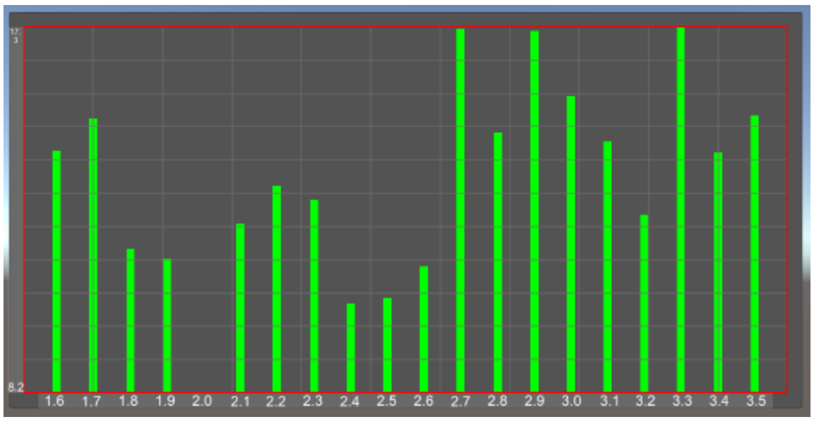
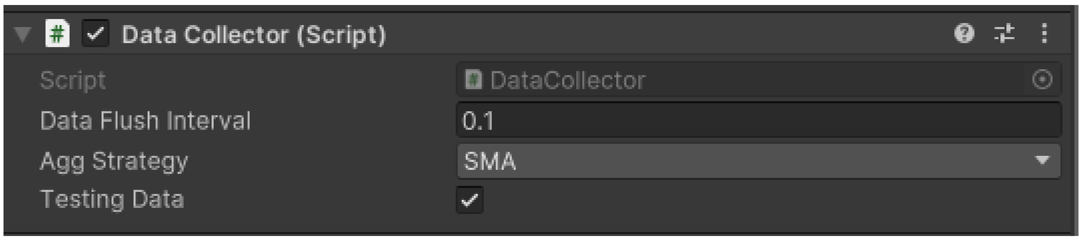


Figure 6.11 Plot drawn using bars representation.

6.4.3 DataCollector

We saw DataCollector component in use in the previous chapter to see some results. This component serves as an entry point to the application. We can also use it for testing purposes, allowing us to feed plots with random testing data. It also can choose from implemented data aggregators, as we can see in the following figure 6.12. We can specify the `DataFlushInterval`, which means how frequently aggregated data points are added to a graph.



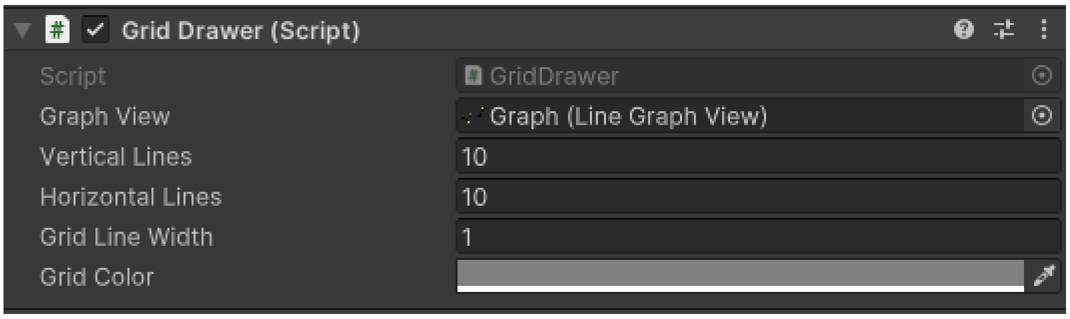
Data Collector (Script)	
Script	DataCollector
Data Flush Interval	0.1
Agg Strategy	SMA
Testing Data	✓

Figure 6.12 DataCollector component.

We can approach this component programmatically as it provides API to feed it with raw data; the method is called `AddRawData`, which takes two parameters. The first is the data point's value on the X-axis, and the second is the value on the Y-axis. When aggregating data points, aggregation is only performed on Y values, and the last X value provided is associated with the aggregated value.

6.4.4 GridDrawer

This component serves as a tool for drawing grids. As we can see in the following figure 6.13, we need to fill in a couple of parameters to work correctly.



Grid Drawer (Script)

Script	GridDrawer
Graph View	Graph (Line Graph View)
Vertical Lines	10
Horizontal Lines	10
Grid Line Width	1
Grid Color	⬜

Figure 6.13 GridDrawer component.

The most important one is to set the reference to our plot drawer. The figure above shows that LineGraphView is assigned so the grid component can have data from the line plot to draw the grid onto. If this parameter is not set up in the inspector, the script tries to find GraphView component on the same game object as itself. Parameters like `VerticalLines` and `HorizontalLines` tell the grid drawer how many lines should be drawn vertically or horizontally. We can also set the line width to make grid lines work with the overall style of our plot. The last parameter we can set up in the inspector is grid color. After we set all parameters to suit our needs, we can get similar results as shown in the following figure 6.14.

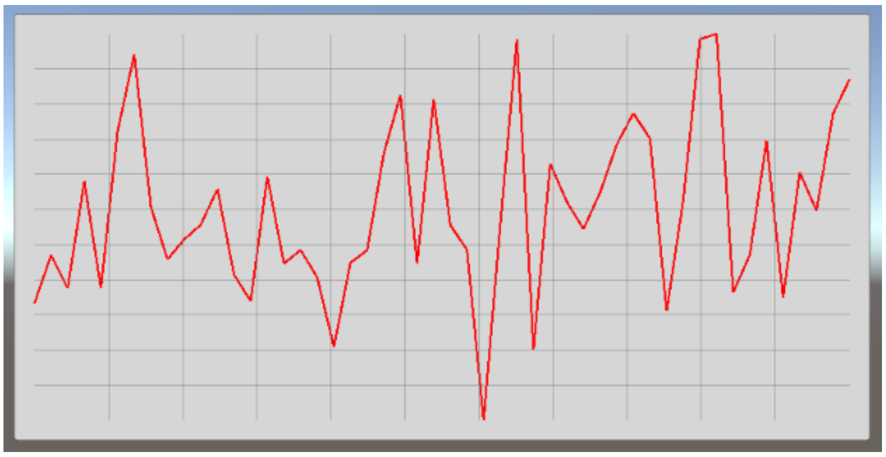
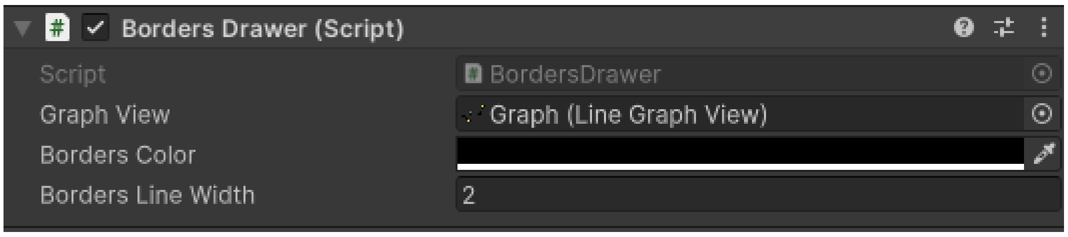


Figure 6.14 Plot drawn by lines with grid component.

6.4.5 BordersDrawer

This component contains functionality to draw borders around our plot. You can define offset from the main UI panel with the background by adjusting margins parameters on the GraphView component. As we see in the following figure 6.15, we need to provide this component with the reference to the GraphView that is currently drawing the plot. If we don't set up any, the script will try to find the GraphView component on the same game object as it is assigned. We can also set up the color and width of the borders.


Borders Drawer (Script)

Script	BordersDrawer
Graph View	Graph (Line Graph View)
Borders Color	
Borders Line Width	2

Figure 6.15 BordersDrawer component.

After adjusting parameters to suit our needs, we can get a similar result to the one shown in the following figure 6.16.

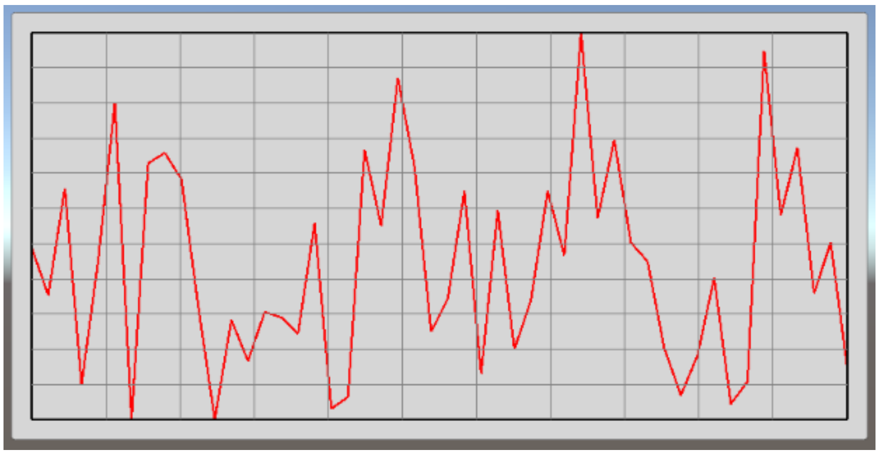
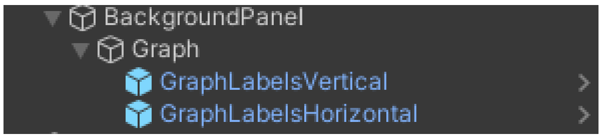


Figure 6.16 Plot drawn by lines with grid and borders components.

6.4.6 GraphLabel

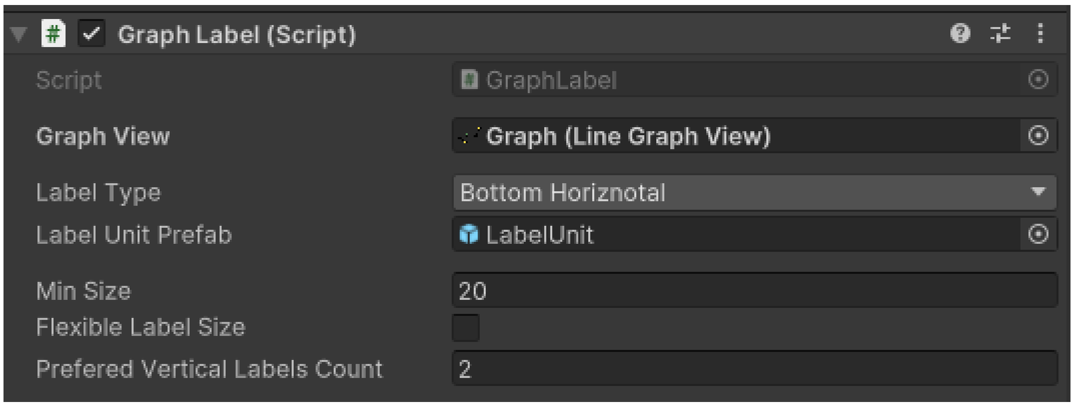
This component provides us with the functionality of the axis. We can locate the predefined axis setup in **Runtime/Prefabs** folder. We can find one prefab for the vertical and one for the horizontal axis. To make the axis work, drag them into the hierarchy under the game object with GraphView component. It is not a must, but it is highly recommended to do it in this way. We should have a hierarchy similar to the one shown in the following figure 6.17.



```
▼ BackgroundPanel
  ▼ Graph
      GraphLabelsVertical
      GraphLabelsHorizontal
```

Figure 6.17 Recommended hierarchy of the game object with axis.

We can locate the GraphLabel component in both prefabs. The component provides us with a couple of options to customize it, as shown in the following figure 6.18.


Graph Label (Script)

Script	GraphLabel
Graph View	Graph (Line Graph View)
Label Type	Bottom Horiznotal
Label Unit Prefab	LabelUnit
Min Size	20
Flexible Label Size	☐
Prefered Vertical Labels Count	2

Figure 6.18 GraphLabel component.

The most important is to set up a reference to the GraphView, as it wouldn't work without it as we need to associate axis values with actual data. We can choose the type of the axis called `LabelType`. We should, however, choose only between the horizontal axis when adjusting the horizontal axis prefab and the vertical axis options when adjusting the vertical one. GraphLabel component needs the reference to the prefab of single label unit as this will serve as a single label for each data point. This label prefab is provided with the package inside **Runtime/Prefabs** folder. This is just a simple game object with Text component on it. We can also specify `MinSize` of the label, which means that for horizontal labels, it will specify minimal height. For vertical labels, it specifies minimal width as the other respective size is adjusted automatically based on a number of data points shown in the plot. When moving margins, we can also decide whether we want the label to resize with the plot. This can be done by checking the box next to the `FlexibleLabelSize` property. The last property to adjust from the inspector is the preferred number of labels on the vertical axis. The default value is set to two, meaning it will show minimal and maximal values in the current data view window. After adding graph label prefabs as proposed above, we can get a similar result to the one shown in the following figure 6.19.

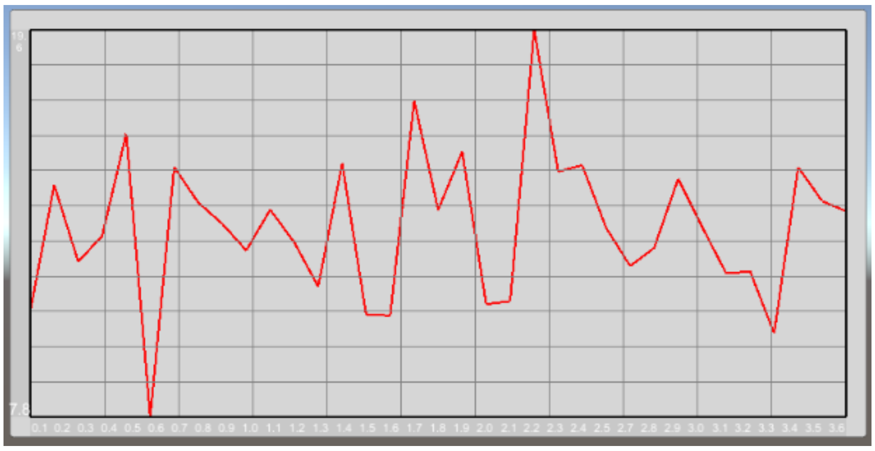


Figure 6.19 Plot is drawn by lines with grid, borders, and axes components.

7 Evaluation

This chapter will evaluate the package we created for real-time data visualization in the Unity engine. We consider primarily how the package stands from the performance point of view and explore limitations in the package.

7.1 Performance testing

One of the goals was to achieve excellent performance, allowing users and developers to use this package with their applications without impacting the overall performance. We focused on great memory management to avoid creating garbage and not force garbage collector to be called often. In the following table, you may see an evaluation of the performance with different amounts of data points at one frame combined with the plotting of a graph using multiple components. The data refresh rate is fixed at 100ms, meaning that after each 100ms, our package aggregates data and provides aggregated data point to be drawn. The aggregation strategy for the whole performance testing is SMA. Performance will be measured in milliseconds to tell us how much CPU time we spend in the worst-case scenarios, meaning we will measure performance drops.

Performance evaluation			
	50 points	200 points	1000 points
LineGraphView	0.17	0.46	2.1
Previous + GridDrawer	0.21	0.5	2.14
Previous + BordersDrawer	0.21	0.52	2.25
Previous + Vertical Axis	0.33	0.64	2.35
Previous + Horizontal Axis	10.2	22.1	85

Table 7.1 Performance evaluation of the package using Unity profiler tool.

We can see that drawing a reasonable number of points and using some graph components doesn't take much computational power. However, we can see that the horizontal axis component performed very poorly. After investigating the problem, we found out that the chosen approach to illustrate each point on the graph as a `Text` component while using `LayoutGroup` is not good at all. We discovered that `LayoutGroup` needs to calculate the positioning and alignment of labels every time we redraw UI, and this operation is rather costly. Vertical axes would be as expensive as horizontal axes as they share the same approach, but we set the fixed label count to two to prevent this performance drop.

We can also see that drawing line graphs, even with 200 points at each frame, is inexpensive and can be used in applications without significant performance drops.

7.2 Limitations

Our package doesn't provide as much functionality as well-established plotting libraries. We can only draw line and bar plots with a couple of simple customization components. Even though this seems like a current limitation, it can be easily expanded to support more customization components.

As we can see in the performance evaluation represented by table 7.1, the performance of the horizontal axis is not good at all, and even for displaying 50 data points, it becomes unusable. We discovered the problem after implementing this component when using a more significant number of data points on the graph.

We haven't been able to test the package commercially yet, and we have not provided it publicly for a larger set of users and developers, so we cannot say with confidence that the package is bug-free. Extensive testing would be required with many use cases that we couldn't come up with. In the real world, many problems with applications and libraries are discovered when they are used by a larger audience and in production.

8 Conclusion

The main goal we set in the section 1.2 was to create a Unity package that provides essential tools for plotting graphs in real-time. In chapter 2, we explored existing plotting libraries to understand use cases and processes that are established and known for users of these libraries. We showed how to create and import a Unity package, which was essential for us to develop and provide a package to work with the Unity engine. We went through analysis in the section 4.3 and offered more solutions to the problems that might occur during implementation.

Our work results in the package with functionality for plotting graphs and a good base for developers to extend the package and its use cases. We rely on the component-based system when constructing plots, as the users are used to using well-known static plotting libraries like *Matplotlib* when using simple commands to add plot components. Our approach in Unity is also more user-friendly, allowing designers and non-programmers to set up their plots directly in the Unity inspector window. We must say we didn't provide as much functionality as we can find in the large packages like *Matplotlib* with a broad user base. Still, we hope this is a great start to help the community of developers who rely on Unity as the primary application development tool.

We created a simple, easily extendible solution for creating different plot visualizations. As a proof of concept, we created two visual representations of the data. We implemented a solution to draw data points and connect them with lines and a bar visualization of data points. We provided plots with the most common components that can be added to enhance plot data visualization. This system is designed to be easily extendible, and any component created similarly can fit into plots immediately. The solution integrates seamlessly with the Unity UI system as we maintain the basic principles on which the Unity engine is built.

8.1 Future work

Our package lacks a large set of customizable components that would enhance the visuals of the plots. In the future, a couple of components could be added or improved based on the following suggestions:

- Add new plot visualization, like a scatter plot.
- Improve axis components to be performant and usable on a large set of data.
- Extend pool of aggregation strategies.
- Add title component.
- Add interactive components like pan and zoom.
- Add the possibility to pause the plot.
- Add the possibility of taking a snapshot of the plot at a specific time.

Also, providing this package publicly and allowing the community to extend it could greatly improve it.

Bibliography

1. Cambridge University Press (ed.). *Cambridge Academic Content Dictionary* [https://dictionary.cambridge.org/dictionary/english/attention-span]. [N.d.]. URL validity: 3.7.2024.
2. Gloria Mark, PhD. *Speaking of Psychology: Why our attention spans are shrinking, with Gloria Mark, PhD* [https://www.apa.org/news/podcasts/speaking-of-psychology/attention-spans]. Ed. by American Psychological association. 2023. URL validity: 3.7.2024.
3. The Matplotlib development team. *Custom hillshading in a 3D surface plot* [https://matplotlib.org/stable/gallery/mplot3d/custom_shaded_3d_surface.html#sphx-glr-gallery-mplot3d-custom-shaded-3d-surface-py]. [N.d.]. URL validity: 5.7.2024.
4. *PYPL PopularitY of Programming Language* [https://pypl.github.io/PYPL.html]. 2024. URL validity: 5.7.2024.
5. Nobledesktop. *Python vs. Excel for Data Analytics* [https://www.nobledesktop.com/learn/python/python-vs-excel]. 2022. URL validity: 5.7.2024.
6. *Introduction to Matplotlib* [https://www.geeksforgeeks.org/python-introduction-matplotlib/]. 2024. URL validity: 5.7.2024.
7. The Matplotlib development team. *Quick start guide* [https://matplotlib.org/stable/users/explain/quick_start.html]. [N.d.]. URL validity: 5.7.2024.
8. Michael Waskom. *An introduction to seaborn* [https://seaborn.pydata.org/tutorial/introduction.html]. [N.d.]. URL validity: 6.7.2024.
9. Michael Waskom. *seaborn.relplot* [https://seaborn.pydata.org/generated/seaborn.relplot.html#seaborn.relplot]. [N.d.]. URL validity: 6.7.2024.
10. Plotly. *Line Plots with plotly.express* [https://plotly.com/python/line-charts/]. [N.d.]. URL validity: 6.7.2024.
11. Plotly. *Getting Started with Plotly in Python* [https://plotly.com/python/getting-started/]. [N.d.]. URL validity: 6.7.2024.
12. Bokeh Contributors. *Bokeh documentation* [https://docs.bokeh.org/en/latest/index.html]. [N.d.]. URL validity: 6.7.2024.
13. Bokeh Contributors. *First steps 1: Creating a line chart* [https://docs.bokeh.org/en/latest/docs/first_steps/first_steps_1.html]. [N.d.]. URL validity: 6.7.2024.
14. The Matplotlib development team. *Animations using Matplotlib* [https://matplotlib.org/stable/users/explain/animations/animations html]. [N.d.]. URL validity: 6.7.2024.
15. The Matplotlib development team. *Introduction to Artists* [https://matplotlib.org/stable/users/explain/artists/artist_intro.html]. [N.d.]. URL validity: 6.7.2024.

16. Plotly. *Intro to Animations in Python* [`https://plotly.com/python/animations/`]. [N.d.]. URL validity: 6.7.2024.

17. Conduct Science. *Games as Research Tools* [`https://conductscience.com/games/`]. [N.d.]. URL validity: 6.7.2024.

18. Mäyrä, Frans; Holopainen, Jussi; Jakobsson, Mikael. Research Methodology in Gaming An Overview. *Simulation & Gaming*. 2012, vol. 43, pp. 295–299. Available from doi: `10.1177/1046878112439508`.

19. Claire Baert. *Citizen Science Games* [`https://citizensciencegames.com/`]. [N.d.]. URL validity: 6.7.2024.

20. Marie Dealessandri. *What is the best game engine: is Unity right for you?* [`https://www.gamesindustry.biz/what-is-the-best-game-engine-is-unity-the-right-game-engine-for-you`]. [N.d.]. URL validity: 6.7.2024.

21. Unity Technologies. *How Unity works with packages* [`https://docs.unity3d.com/Manual/upm-overview.html`]. [N.d.]. URL validity: 6.7.2024.

22. Unity Technologies. *Install a UPM package by name* [`https://docs.unity3d.com/Manual/upm-ui-quick.html`]. [N.d.]. URL validity: 6.7.2024.

23. Microsoft. *Fundamentals of garbage collection* [`https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals`]. [N.d.]. URL validity: 6.7.2024.

24. Unity Technologies. *Line Renderer component* [`https://docs.unity3d.com/Manual/class-LineRenderer.html`]. [N.d.]. URL validity: 6.7.2024.

25. Unity Technologies. *GL* [`https://docs.unity3d.com/ScriptReference/GL.html`]. [N.d.]. URL validity: 6.7.2024.

26. Unity Technologies. *Graphic* [`https://docs.unity3d.com/2019.1/Documentation/ScriptReference/UI.Graphic.html`]. [N.d.]. URL validity: 6.7.2024.

27. Arpaci-Dusseau, Remzi H.; Arpaci-Dusseau, Andrea C. (2014). *Operating Systems: Three Easy Pieces [Chapter: Condition Variables, figure 30.13]* [`https://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf`]. Ed. by Arpaci-Dusseau Books. [N.d.]. URL validity: 6.7.2024.

28. Unity Technologies. *RequireComponent* [`https://docs.unity3d.com/ScriptReference/RequireComponent.html`]. [N.d.]. URL validity: 6.7.2024.

29. Reddy, M. *API Design for C++*. Elsevier Science, 2011. isbn 9780123850041. Available also from: `https://books.google.cz/books?id=IY29LylT85wC`.

30. NumFOCUS, Inc. *pandas.DataFrame* [`https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html`]. [N.d.]. URL validity: 6.7.2024.

31. Hashemi-Pour, Cameron. *What is a user interface (UI)* [`https://www.techtarget.com/searchapparchitecture/definition/user-interface-UI`]. Ed. by Churchville, Fred. 2024. URL validity: 5.7.2024.

32. Unity Technologies. *Package manifest* [`https://docs.unity3d.com/Manual/upm-manifestPkg.html#required`]. [N.d.]. URL validity: 6.7.2024.

33. UNITY TECHNOLOGIES. *Package layout* [`https://docs.unity3d.com/Manual/cus-layout.html`]. [N.d.]. URL validity: 6.7.2024.

List of Figures

2.1 Terrain 3D Plot. Source: Matplotlib.org [3] 11
2.2 Plotting Libraries user base (Obtained from their Github repositories) 13
2.3 Components of a *Matplotlib* Figure. Source: Matplotlib.org [7] . . 15
2.4 Basic line graph using Matplotlib. 17
2.5 Basic bar graph using Matplotlib 18
2.6 Plot representing data of the customer [9]. 19
2.7 Example interactive plot with hover over the specific point. Source: plotly.com [10] 21
2.8 Plotting Libraries user base (Created using Plotly) 22
2.9 Line plot created by using *Bokeh* [13]. 24
2.10 Animated bar plot using *Plotly*, generated by code 8. Source: plotly.com [16] 26

3.1 Proccess of updating packages in Unity. Source: docs.unity3d.com [21] 30
3.2 Unity package manager window in the editor. 30

4.1 Components assigned onto game object in Unity. 37

5.1 Diagram representing relations between parts of our package architecture. 42
5.2 Line segment constructed from two triangles and four vertices. . . 45
5.3 Unity Test Runner window when every test passes successfully. . . 49

6.1 Running GraphScene in play mode. 50
6.2 Hierarchy of game object in the GraphScene. 51
6.3 Creating canvas in the hierarchy. 51
6.4 Creating panel in the hierarchy under canvas. 52
6.5 Adjusting anchors on RectTransform component. 53
6.6 Components onto game object after adding LineGraphView. 53
6.7 Graph game object after adding DataCollector component. 54
6.8 Basic line chart in game mode. 54
6.9 LineGraphView component. 55
6.10 BarGraphView component. 55
6.11 Plot drawn using bars representation. 56
6.12 DataCollector component. 56
6.13 GridDrawer component. 57
6.14 Plot drawn by lines with grid component. 57
6.15 BordersDrawer component. 58
6.16 Plot drawn by lines with grid and borders components. 58
6.17 Recommended hierarchy of the game object with axis. 58
6.18 GraphLabel component. 59
6.19 Plot is drawn by lines with grid, borders, and axes components. . 59

A.1 Recommended package layout. Source: docs.unity3d.com [33]. . . 70
A.2 Example assembly definition file in Unity project explorer. 70

List of Tables

7.1 Performance evaluation of the package using Unity profiler tool. . 60

List of Abbreviations

API An application programming interface (API) is a way for two or more computer programs or components to communicate with each other [29].. 12, 18, 47

DataFrame Two-dimensional, size-mutable, potentially heterogeneous tabular data [30].. 18, 19

PYPL The PYPL PopularitY of Programming Language Index is created by analyzing how often language tutorials are searched on Google: the more a language tutorial is searched, the more popular the language is assumed to be. It is a leading indicator. The raw data comes from Google Trends [4].. 11

UI The user interface (UI) is the point of human-computer interaction and communication in a device. This can include display screens, keyboards, a mouse and the appearance of a desktop. It is also how a user interacts with an application or a website, using visual and audio elements, such as type fonts, icons, buttons, animations and sounds [31].. 11, 12, 13, 36

A Attachments

A.1 Creating Unity package

To create a Unity package, the following steps must be reproduced :

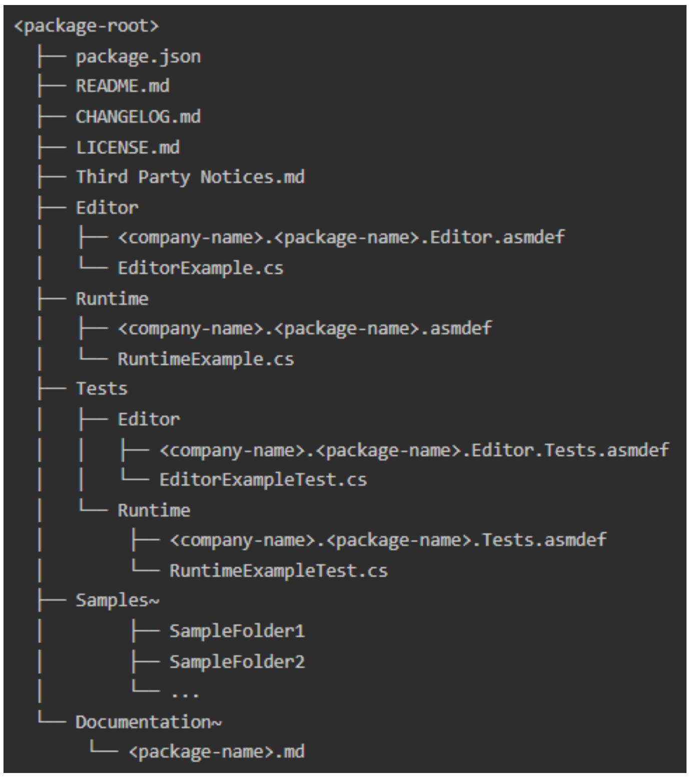
1. First, we need to create an empty package. The easiest way is to use your computer's file manager to create a folder for your package.

2. Create a file named **package.json** in the root of the created folder. The required components to fill are **name**, which has to be in format *com.[company-Name].[package-Name]* such as *com.randomcompany.mypackage*. Another required component is **version**. The version string must follow package version number convention in the form **MAJOR.MINOR.PATCH**, such as **5.1.2**. The other fields that can be filled in **package.json** file can be found in official Unity documentation [32]. The following code 10 represents an example manifest of the package for better understanding.

Program 10 Example package manifest [32]

```
{
    "name": "com.[company-name].[package-name]",
    "version": "1.2.3",
    "displayName": "Package Example",
    "description": "This is an example package",
    "dependencies": {
        "com.[company-name].some-package": "1.0.0",
        "com.[company-name].other-package": "2.0.0"
    },
    "author": {
        "name": "Unity",
        "email": "unity@example.com",
        "url": "https://www.unity3d.com"
    }
}
```

3. Layout of the package must follow the package layout convention for Unity packages. As shown in the following figure A.1.



```
<package-root>
  ├── package.json
  ├── README.md
  ├── CHANGELOG.md
  ├── LICENSE.md
  ├── Third Party Notices.md
  ├── Editor
  │   ├── <company-name>.<package-name>.Editor.asmdef
  │   └── EditorExample.cs
  ├── Runtime
  │   ├── <company-name>.<package-name>.asmdef
  │   └── RuntimeExample.cs
  ├── Tests
  │   ├── Editor
  │   │   ├── <company-name>.<package-name>.Editor.Tests.asmdef
  │   │   └── EditorExampleTest.cs
  │   └── Runtime
  │        ├── <company-name>.<package-name>.Tests.asmdef
  │        └── RuntimeExampleTest.cs
  ├── Samples~
  │        ├── SampleFolder1
  │        ├── SampleFolder2
  │        └── ...
  └── Documentation~
       └── <package-name>.md
```

Figure A.1 Recommended package layout. Source: docs.unity3d.com [33].

4. The layout must have assembly definitions files if the package includes code. Assembly definition files are the Unity equivalent to a C# project in the .NET ecosystem. Those files can be created in the Unity by following steps:
   - Right-click on the mouse while in the Project tab.
   - Find *Create* in the drop-down menu
   - Click on *Assembly Definition*

   It will create the file as shown in the following figure A.2.

Figure A.2 Example assembly definition file in Unity project explorer.

5. Add necessary tools, code, and assets into your package folder.
6. As to follow good practice, create tests to test your code inside the package. Test scripts must be stored in the *Tests* folder as shown in the figure A.1. Place your editor tests in the **Editor** folder and your runtime scripts in the **Runtime** folder.
7. To add samples such as sample scenes, those should be placed in the **Samples** folder. Note that when the folder is created with its name followed

by character ' ', Unity ignores this file to create a .meta file and to show it in the project tab. As Unity might recommend this, I found it rather unpleasant when importing and exploring the package; this **Samples** folder might be easily missed, even though it is possible to interact and import samples through the package manager window.

8. Good practice is to create **Changelog** file for the user to track recent changes more quickly.
9. If needed **License** and **ThirdPartyNotices** filed can be created.
10. The package documentation might be one of the most crucial things that must be created. Even though it is optional, I strongly recommend it, as users will struggle to use your package without proper documentation.
11. To easily share your package. Open it in your computer's file explorer and compress the root folder of your package. Anyone with Unity can import it using *Unity Package Manager*.