PRIFYSGOL
BANGOR
UNIVERSITY

School of Computer Science
College of Physical & Applied Sciences

Modular Analytics for Unity Games

Spencer Alexander Rose

Submitted in partial satisfaction of the requirements for the
Degree of Bachelor of Science
in Creative Technologies

Supervisor Llyr Ap-Cenydd

May, 2016

Statement of Originality

The work presented in this thesis/dissertation is entirely from the studies of the individual student, except where otherwise stated. Where derivations are presented and the origin of the work is either wholly or in part from other sources, then full reference is given to the original author. This work has not been presented previously for any degree, nor is it at present under consideration by any other degree awarding body.

Student:

Spencer Alexander Rose

Statement of Availability

I hereby acknowledge the availability of any part of this thesis/dissertation for viewing, photocopying or incorporation into future studies, providing that full reference is given to the origins of any information contained herein. I further give permission for a copy of this work to be deposited with the Bangor University Institutional Digital Repository, and/or in any other repository authorised for use by Bangor University and where necessary have gained the required permissions for the use of third party material. I acknowledge that Bangor University may make the title and a summary of this thesis/dissertation freely available.

Student:

Spencer Alexander Rose

Abstract

This project seeks to develop a set of tools for the Unity game engine which provide a flexible interface for gathering *analytics* data - statistical information about how users interact with a game. At present, the Unity engine provides some built-in functionality for the collection of user data, as well as an online interface for reviewing collected data. However, Unity's existing functions are inflexible, and require developers to hard-code the variables which will contribute to the analytics collection process. The tools resulting from this project seek to remove this requirement, meaning that developers can specify and adjust the data collection targets without editing any script files.

These tools have been developed in the form of re-usable *components* for the Unity engine, which were evaluated within a sample game project developed by Unity Technologies (the Unity *Survival Shooter* demo). These components were developed following an investigation into the existing analytics functionality within the Unity engine, identifying a way in which the functions of Unity's analytics system could be combined with a *component-based* design - an approach commonly adopted within the Unity editor and development environment.

The result is a working prototype of these components. Some operational and performance limitations may preclude practical applications for this set of tools in their current state, and these have been identified and discussed.

Conventions

Within this report, a convention is used to highlight .NET classes, interfaces, or namespaces (which are printed in `green`), Unity engine classes (printed in `orange`), and classes developed within this project (printed in `magenta`). When referring to elements of code, such as keywords or method names, a `monospaced` font is used. Additionally, variable names are ‘`enclosed`’ in single quotes.

Contents

List of Figures

Chapter 1

Introduction

Remotely collecting data about how users play and interact with games is becoming a common process among game developers [1, pp. 4-6], [2, pp. 231-232]. This data can be of benefit to a wide range of staff within a game development studio, including designers, marketing personnel, and management [1, pp. 42-46] - roles which may involve little to no knowledge of programming [3, p. 1], [4]. The overall process of collecting this data is referred to as *analytics*, while the data itself is referred to as *telemetry* - a convention described by El-Nasr et al. [1, p. 5], and adopted within this report.

Game developers often utilize *game engines* - software suites which provide a range of generic tools and building blocks which can be applied to the development of a variety of games [5]. The *Unity* engine is a popular choice among game developers, taking 45 percent of the market share for game engines [6].

The Unity engine features an online analytics service, and a series of functions which developers can use to collect telemetry data. This is a relatively new service - while the first version of the Unity engine was released in 2005 [7], the analytics tools were added in 2015 [8].

Many functions in Unity are implemented using a *component-based* design - generic elements such as lights, cameras, and audio sources are available within the Unity editor as customizable *components*. This is advantageous for developers without programming experience, as they can create and manipulate scenes and objects without having to write code or create new script files. At the time of writing, however, the Unity editor does not feature

any components which deal with analytics collection behaviours. Developers who wish to incorporate analytics collection into their projects must append code into scripts at the specific point where data is to be collected.

As a result, this project seeks to determine whether a set of analytics collection components could be an efficient and flexible alternative to Unity's current method of gathering data for analytics applications. To realize this aim, the following objectives were developed:

1.1 Objectives

- Explore Unity's existing analytics system and functions, and investigate how the .NET Framework can inspect variables and types while an application is running.

- Develop a prototype set of *flexible* and *easy to use* components for the Unity engine, which provide an interface for analytics data collection.

- Implement and evaluate the components within the environment of a preexisting Unity application.

- Evaluate the computational efficiency of the developed components against Unity's current data collection method.

- Time permitting, evaluate the components in a wider range of projects, and perform a user study.

Chapter 2

Background

This section discusses several topics relating to the project and its implementation:

- The subject of analytics is covered in section 2.1, and provides an overview of analytics processes, their role in games, and associated concepts.

- The .NET Framework, Mono tools, and specific points relating to the C# language are discussed in section 2.2.

- The Unity engine is discussed in section 2.3. An overview of existing analytics tools for Unity is also provided.

- The importance of real-time performance for gaming applications is briefly discussed in section 2.4.

2.1 Analytics, Telemetry, and Metrics

Since this project deals with an *analytics* system, it's important to convey an understanding of analytics and associated terminology. El-Nasr et al. describe analytics as "the process of discovering and communicating patterns in data" [1, p. 14]. They go on to note that the study of analytics is not simply concerned with the acquisition of data, but rather the analysis and interpretation of acquired data [1, p. 14]. The motives for collecting such data can be widespread, but are generally business-related. Supporting business decision-making through the use of data (or the analysis of data) can lead to more intelligent decisions, predictions, or business goals; El-Nasr et al. describe this as becoming "data-driven" [1, p. 14].

The term *telemetry* (or, interchangeably, *analytics data*) is used to distinguish the collected data from the actual data collection processes [1, p. 5, pp. 16-17]. A separate term, *metrics*, is used to refer to the "objective" which the telemetry data is expected to measure [1, p. 5]. The distinction is that telemetry is simply a representation of data (such as an integer or a vector), while a metric gives meaning to the data; by this definition, the same piece of telemetry data can provide different meanings depending on the context created by a metric [1, pp. 17-19].

An analytics process and procedures taken to collect telemetry data are variable - El-Nasr et al. describe a series of eight steps involved in developing an analytics process, which are based on existing processes of "knowledge discovery" used in data mining [1, pp. 35-37, pp.210-212]. In this project, an *interface* to an existing analytics process has been developed, so we are primarily concerned with the first and second steps of the procedure described by El-Nasr et al. The first step, 'attribute definition', deals with identifying and selecting which data should be gathered, and the point at which the data should be collected. The second step, 'data acquisition', deals with the collection of data using the specifications established during the 'attribute definition' stage. A few key elements can be identified from these two steps - a data collection *target*, a *trigger* or *event* which starts the 'data acquisition' procedure, and the actual acquisition procedure itself.

As described in chapter 1, this report adopts the definitions of analytics, telemetry, and metrics provided by El-Nasr et al. The purpose of doing so is to avoid confusion between *telemetry* (the data) and *metrics* (a measure for interpreting the data), as well as distinguishing between the collected *telemetry* data and the *analytics* processes themselves.

2.1.1 Analytics in Gaming

Overview

Game developers have begun to use analytics collection processes in their products as a way to learn from their players [1, pp. 3-6], [2, pp. 231-232],

or to support internal development processes [1, pp. 111-112]. Collection of player data is particularly useful for developers working on *free to play* (F2P) products. These are games or applications which are distributed for free, generating revenue through *microtransactions* - virtual items or currency which players can purchase with real-world money [9, p. 72]. Relying on analytics data can help identify the actions and purchases which players make within a game, which in turn can be used to influence sales, determine what types of content should be added to the game, and so on [1, p. 32], [9, p. 109-110].

Using analytics data to influence sales might be mostly applicable to F2P games, but the processes of gathering telemetry data and using metrics to make informed decisions are applicable regardless of game, genre, or business model. The data, and associated metrics, can be beneficial to many parties involved in a game's development [1, pp. 42-50]. Game designers, for instance, can use metrics to help in determining how a game could be improved or expanded upon. Management can use metrics to decide whether it would be more valuable to continue supporting a game, or to begin development of a new project. Programmers can look at metrics relating to a game's computational performance to identify areas in need of optimization or improvement. Even players can benefit from the data they provide. Telemetry data (either provided by developers, or gathered through systems implemented by players) can be useful in refining or developing new strategies, or in changing the ways in which players interact with a game [1, p. 50, p.417].

Implementations

The range of use-cases and applications for analytics processes in games are extensive, given the many different types and genres of games. To illustrate the variety and various types of analytics collection procedures used in commercial game software, examples from industry are discussed in this section.

A 2008 paper by Microsoft researchers describes how a telemetry collection system helped developers working on the games ‘Halo 2’ and ‘Shadowrun’ [10, pp. 443-450]. Specifically, their system is employed during player testing to identify areas of the game which need improvement - although the data is intended to serve developers, it is a *player-facing* system [1, p. 112]. Their process is built around the recording of “sequences of events” [10, p. 445] to gain a contextual understanding of the telemetry data [10, p. 445]. Reviewing every action a player performed during a game would be too cumbersome and time-consuming, so there is reliance on key *events* (discussed in section 2.1) which begin the data collection process.

A scenario described by the authors demonstrates how their system was able to help developers. Analysis of player data during ‘Halo 2’ tests identified that players were dying frequently at a certain point within the game. The telemetry data helped to identify this point, but not the cause of player deaths. By reviewing the sequence of events, developers noted that the deaths were caused by a specific type of enemy character. And by exploring data associated with the event, the developers were able to identify the specific enemy behaviour which caused player deaths [10, p. 448]. The authors note that the system has been “invaluable” [10, p. 451], and also note that it was incorporated into additional games and software products being developed by Microsoft [10, p. 451].

While the Microsoft system is concerned with gathering player data, a *developer-facing* system built for game studio Bioware is concerned with the opposite - it deals with collection of data from developers [1, p. 111-112]. The goal of such a system is not necessarily to improve the player experience of a game, but to improve internal development processes - the data is primarily of use to management and production staff. By reviewing data collected through this system, developers were able to identify internal processes which were negatively affecting productivity [1, p. 119]. As a result, development efforts could be refocused, and programmers could be assigned to tasks which would have “maximum impact” [1, p. 119].

Valve Software is a game development company which uses “data-driven” decision making processes [11]. Examples from two of their games (‘Team Fortress 2’ and ‘Counter-Strike: Global Offensive’) exemplify how data is used within the game development process, and how the data can also benefit players.

In ‘Team Fortress 2’, telemetry collection functions gather personalized, statistical data about players. Examples include the total time a player has spent within the game, the character with which a player has caused the most damage to other players, and so on. An interface is provided within the game for players to review this data, as depicted in figure 2.1. Aggregating this data might be useful to developers - to determine whether a majority of users prefer playing as one character over the others, as an example. The individual data can also be of interest to each player. Friends can compare their statistics, or players can set personalized challenges for themselves (beating their current records, for instance).

Figure 2.1: Player statistics display from ‘Team Fortress 2’ [12]

Valve notes that a large volume of data is collected from ‘Counter-Strike’ players [13]. Much like Microsoft’s system, discussed previously, Valve combines individual pieces of telemetry data to gain a greater, contextual understanding of game events. In the case of ‘Counter-Strike’, an example metric is the tracking of a player’s location within the game world once the player fires their weapon [14]. These individual positions are aggregated,

and displayed using a *heat map* - a graphical representation of the game world, colored differently based on the volume of overlapping data points [14]. For developers, the resulting maps identify areas which players traverse most often. From a player's perspective, the maps might identify the most 'dangerous' parts of a game world.

All of the systems discussed in this section share common elements which were discussed near the end of section 2.1. There is a dependence on the *identification* of data items to track, and *events* which cause the data to be captured and recorded. Also illustrated through these examples is the diversity in *stakeholders* [1, pp. 42-51] which depend on analytics to influence their decision making processes, including game designers, producers, programmers, and players.

2.2 .NET, Mono, and C#

.NET (dot net) is a software development *framework* - a package providing re-usable software components for programmers. The Unity engine depends on an implementation of the .NET Framework, so it is important to present a basic understanding of .NET, and some of its specific features. Microsoft, developers of the .NET Framework, break it down into two parts - the *common language runtime* (CLR) and the *framework class library* (FCL) [15].

2.2.1 .NET CLR and FCL

The FCL provides the 'reusable' elements of the .NET Framework, consisting of many classes and types for programmers to use [16]. These classes and types are organized within *namespaces* - a construct used to group and qualify classes or types [17].

The CLR deals with the execution of code written for the .NET Framework. Compilers translate source code, written in any language supported by .NET, into *Intermediate Language* (IL), or *managed code* [18], [19, p. 6]. The IL code is stored in an *assembly* [20, pp. 27-28] which can be processed by the CLR, resulting in *native* code which a target device can execute [21].

2.2.2 Mono

Microsoft's CLR for the .NET Framework requires a Windows operating system in order to execute [20, pp. 4-5]. As a result, alternatives have been developed so that programs built upon the .NET Framework can be executed on a wider range of devices. One such alternative is *Mono* - an open-source implementation of the .NET Framework. Unlike Microsoft's .NET CLR, Mono's runtime is able to execute IL code on non-Windows operating systems and a range of hardware, including mobile devices [22].

The Unity engine is capable of building executable files for several different operating systems and hardware configurations (referred to as *platforms*) [23]. This is possible, in part, due to Unity's implementation of Mono for executing user-created scripts [24]; dependence on the Microsoft CLR would limit games built with Unity to Windows-based systems.

2.2.3 Metadata and Attributes

As discussed in section 2.2.1, a compiler converts code for .NET applications into IL. The compiler also generates *metadata* about the code it processed, and stores this with the IL code [25], [19, pp. 38-40]. The .NET runtime uses metadata when executing code [20, pp. 19-22], but programmers also have access to this metadata through *reflection*, discussed in section 2.2.6.

In addition to compiler-generated metadata, users may specify *attributes* relating to their code [26]. These can be retrieved through reflection, just like metadata. The FCL contains some standard attributes [27], but custom attributes can be specified by creating a new class which inherits from the `System.Attribute` class [26]. Unlike metadata, which is generated by a compiler, attributes are explicitly specified by a programmer.

As described in section 4.2.3, a custom attribute is used in this project to describe methods which support the analytics collection process.

2.2.4 C#

All code in this project was written in C# (C sharp). C# was selected due to its design origins (as a language built around the development of .NET applications) [28], and its integration into the Unity engine as one of two supported languages - the other being UnityScript, a derivative of JavaScript [29]. Since Unity can directly build an *assembly* from C# or UnityScript files, a separate step in the build process can be skipped [30].

Types, Members, Fields, and Properties

It is worth briefly discussing *types* and *members*, since they will be discussed often in relation to the project's implementation. C# is *strongly typed*, so every variable or object is associated with a specific *type* [31]. *Members* encompass methods or variables defined within a class [32]. *Fields* and *properties* are both members which represent variables, but properties have a fundamental difference - their definitions can contain integrated *accessor* methods [33]. A typical convention used throughout the code in this project is the use of a field declared as `private` (the *backing field*), and a corresponding `public` property. Using a backing field, in this manner, allows the property's data to be *serialized* by the Unity engine, which is discussed in section 2.2.5. In the example given in figure 2.2, the '`eventName`' field is set and retrieved using the accessors belonging to the '`EventName`' property.

```
1      private string eventName;
2
3      public string EventName
4      {
5          get { return eventName; }
6          set { eventName = value; }
7      }
```

Figure 2.2: Private field and public property with accessors

2.2.5 Serialization

By *serializing* an object, it is converted into a sequence of bytes, and by *deserializing* an object, byte sequence is converted to a representation of an object [19, p. 611]. Richter provides a few sample use-cases for serialization, such as saving and later restoring an application's state, or sending objects' state through a network to another process [19, p. 611]. In the .NET Framework, types are defined as serializable by applying the `[Serializable]` attribute [19, p. 618], [34]. The use of serialization within the Unity engine is explored further in section 2.3.4.

2.2.6 Reflection

Types within the `System.Reflection` namespace provide a means of inspecting metadata associated with types in a given assembly [35]. By using reflection to inspect an assembly, programs can access information about types which may have been unknown when the program was first compiled [19, p. 589]. Using reflection to interact with types in this manner is a form of *late binding* [19, p. 589] since a listing of the types is loaded and examined during the application's runtime.

This project primarily relies upon the `MemberInfo` class for inspecting metadata [36]. Several types which inherit from the `MemberInfo` class (such as `PropertyInfo`, `FieldInfo`, and `MethodInfo` [36]) are also used extensively.

A simple example of property inspection using reflection is given in figure 2.3. The program's output is shown in figure 2.4.

```
1       object o = "string";
2       Type t = o.GetType();
3       PropertyInfo[] pi = t.GetProperties();
4       Console.WriteLine("The type is " + t.FullName);
5       foreach(PropertyInfo p in pi)
6       {
7           Console.WriteLine("property " + p.Name);
8       }
```

Figure 2.3: Retrieving a type's properties with reflection

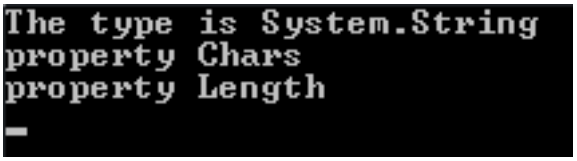

Figure 2.4: Output from figure 2.3 code

All types in C# inherit from `object` [37], so an `object` can be used as a container for various types. In this case, the object 'o' was assigned a string literal. The object's type (identified by using the `GetType` method) was determined to be of `System.String`. The `GetProperties` method (which returns an array of `PropertyInfo`) is called on the object's `Type`. In this case, two properties were returned - `Chars` and `Length`. The C# documentation for `System.String` confirms that these properties belong to the string class [38].

```
1  public object GetValueExample(AnalyticsMemberInfo m)
2  {
3      Type t = Type.GetType(m.QualifiedName);
4
5      // Unity specific - retrieve Component of type 't' attached to this object
6      Component c = gameObject.GetComponent(t);
7
8      if (m.MemberType == MemberTypes.Field) // member is a field
9      {
10         FieldInfo fi = c.GetType().GetField(m.Name);
11         return fi.GetValue(c).ToString();
12     }
13     else if(m.MemberType == MemberTypes.Property) // member is a property
14     {
15         PropertyInfo pi = c.GetType().GetProperty(m.Name);
16         return pi.GetValue(c).ToString();
17     }
18     else
19         return null;
20 }
```

Figure 2.5: Retrieving a field or property value with reflection

The example in figure 2.5 demonstrates the acquisition of a value without the use of a direct reference (e.g., writing `object.value` to access the 'value' property belonging to `object`). Instead, reflection is used to obtain a `MemberInfo` object from a *string* representation of a field or property's name. Given the `MemberInfo` object describing the property or field to be examined, the `GetValue` method can be provided with an object reference to a matching `Type`. In this case, the object reference is a Unity-specific type, a `Component`. With a reference to an object, and a `MemberInfo` describing the property or

value to be inspected, the value can be acquired *without* having to write `object.value` directly.

The examples from figures 2.3 and 2.5 illustrate two of the primary ways in which reflection enables data collection within this project. This is described in further detail throughout chapter 4.

2.3 Unity Engine

As discussed in section 1, Unity is a *game engine* - software which hastens the process of creating a game by providing developers with a set of pre-built functions and tools [5]. In this case, the Unity editor provides developers with existing systems to handle animation, rendering, audio, and physics calculations [39]. By using a game engine, developers can focus their efforts on writing code specific to their project, rather than re-engineering 'generic' systems which deal with rendering, audio, and so on. Unity Technologies, developers of the Unity engine, also operate the Unity "asset store", a marketplace of user-created content and tools [40].

Unity's native language support was previously mentioned in section 2.2.4; the Unity editor can compile and build assemblies for code written in C# or 'UnityScript' - a Javascript-like language developed for the Unity engine [29]. However, Unity is capable of executing code written in other .NET compatible languages, if a '.DLL' file containing IL code can be generated [30].

2.3.1 Base Classes

Game environments in Unity are contained within *scenes*. Implementing functionality within a scene depends on two base classes from the `UnityEngine` namespace - the `GameObject` and `Component`. Unity's documentation describes `GameObjects` as a container for `Components`, where the components provide specific functionality or behaviour [41], [42]. Every component is associated with a script (deriving from the `MonoBehaviour` class [43]), which defines its behaviour or functionality [29]. One of the objectives

set for this project was the implementation of components to handle telemetry collection, as noted in section 1.1. As a result, this project includes a `Component` script which handles this behaviour, discussed further in section 4.1.2.

2.3.2 GameObjects

Instances of the `GameObject` class represent containers for `Components`[41], discussed in section 2.3.3. Every instance of a `GameObject` is created by a user in the Unity editor, or instantiated via code within a script [44]. Every instance contains a `Transform` component [45], representing the object's position within the game scene, and any transformations (scale or rotation).

`Components` attached to a `GameObject` are capable of referencing other `Components` attached to the same object [46]. Furthermore, the objects may be arranged into *hierarchies* (see figure 2.6), and `Components` may explore the entire hierarchy tree to access other `Components` [46], [47].

Figure 2.6: Unity editor hierarchy window, showing a series of parent and child GameObjects

An empty `GameObject` is depicted in figure 2.7. It is not possible to remove the `Transform` component [45] - the object always maintains a position and transformations within the currently loaded scene. In the

context of this project, we are mostly concerned with accessing existing `GameObjects` and attaching a custom `Component`, rather than setting up and controlling interactions between objects. Nevertheless, it is important to have an understanding of the `GameObject` class, and its relationship with the `Component` class, since this forms the basis of any Unity project.

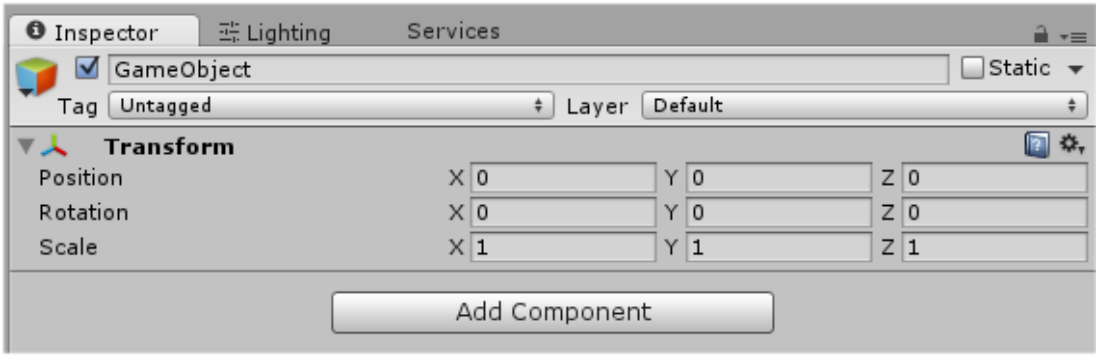

Figure 2.7: Unity editor inspector window, examining a newly created GameObject

2.3.3 Components

The `Component` class has been mentioned in previous sections. Section 2.3.1 highlights the relationship between `Components` and script files, and section 2.3.2 discusses the relationship between the `GameObject` and `Component` classes.

Throughout this report, reference is made to *component-based* design. In the context of the Unity engine, this is the implementation of various functions and tools as `Components`. An example of such a function has already been provided in figure 2.7 - the `Transform` component provides a means of representing and adjusting translations and transformations of a `GameObject` [48]. A more practical example of `Components` is provided in figure 2.8, visually demonstrating behaviour provided by the `Light` and `Halo` `Components` which are included with Unity. In this example, the `Light` simulates the casting of light on other scene objects, while the `Halo` provides a visible 'glow' effect.

Generally, each `Component` is concerned with performing a task or procedure related to a specific `GameObject` [42]. Therefore, the component-based design could be thought of as promoting the *single responsibility principle* [49, p.

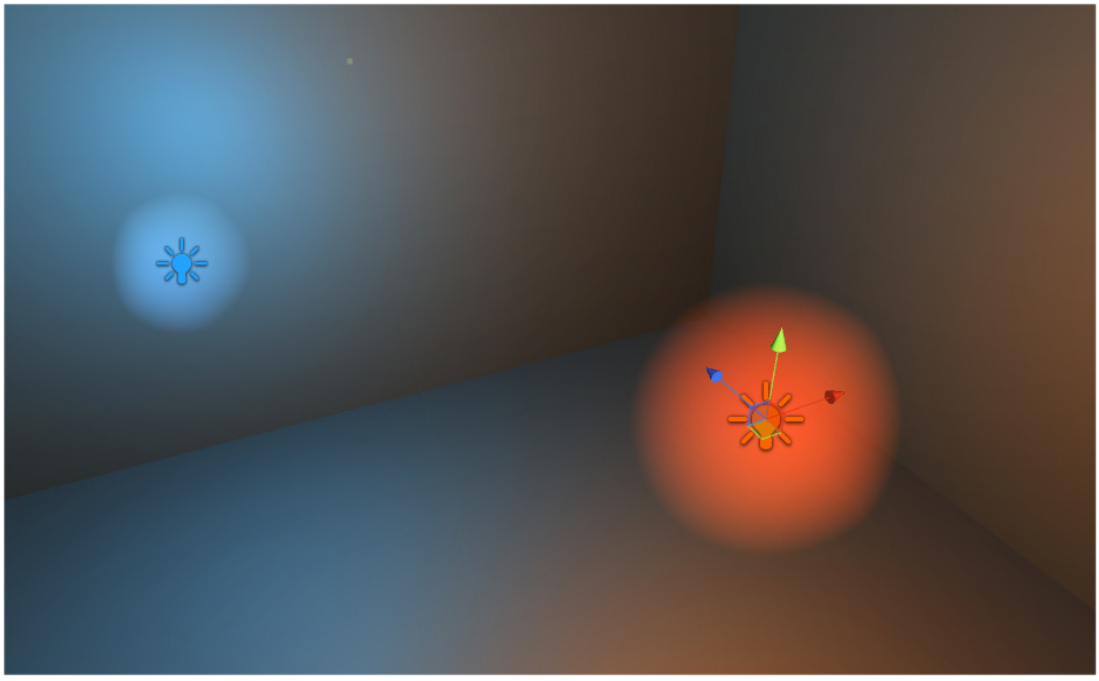

Figure 2.8: Unity scene demonstrating behaviour of `Light` and `Halo` `Components`.

169], which dictates that classes should be responsible for one aspect of a program's behaviour. However, since `Components` may be responsible for multiple behaviours, it may be better to view them as promoting *reusable code*, instead of an embodiment of the single responsibility principle. In this project, and throughout this report, the term *modular* refers to the 'reusable' nature of `Components`.

The Unity editor enables manipulation of `public` variables contained within `Component` scripts [50]. This is accomplished by using the *inspector* window in the Unity editor, discussed throughout section 2.3.5.

2.3.4 Serialization in Unity

Unity relies on serialization to save and load the state of `GameObjects` and `Components` within a scene, to store settings and configurations within the editor application, and so on [51]. Unity's serialization system applies its own restrictions on what can and cannot be serialized; specifically, Unity is only capable of serializing *fields*, and will only serialize fields with the `private` access modifier if the `[SerializeField]` attribute has been applied [52]. Furthermore, the fields may not be declared as `static`, `const`, or `readonly`. Unity is incapable of serializing properties [52], so in this project, a backing

field (discussed in section 2.2.4) is always used when a property's data must be serialized.

Additionally, Unity will only serialize certain types [52], listed in figure 2.9.

- `enums`
- `structs`
- Classes which inherit from `UnityEngine.Object`.
- Basic data types - `int`, `float`, `string`
- Unity data types and structs - `Vector3`, `Vector4`, `Matrix4x4`
- Arrays of serializable types
- Lists of serializable types

Figure 2.9: List of types which Unity engine can serialize [52]

2.3.5 Editor Interface

The Unity editor has several key windows, depicted in figure 2.10. The *hierarchy* window lists all `GameObjects` contained within the currently loaded scene, while the inspector provides a graphical interface for adjusting `GameObjects` and any attached `Components`. The project panel displays a list of any 'assets' associated with the project, and the scene view displays a visual representation of the currently loaded scene. There is also a console window which displays output from scripts and the editor application.

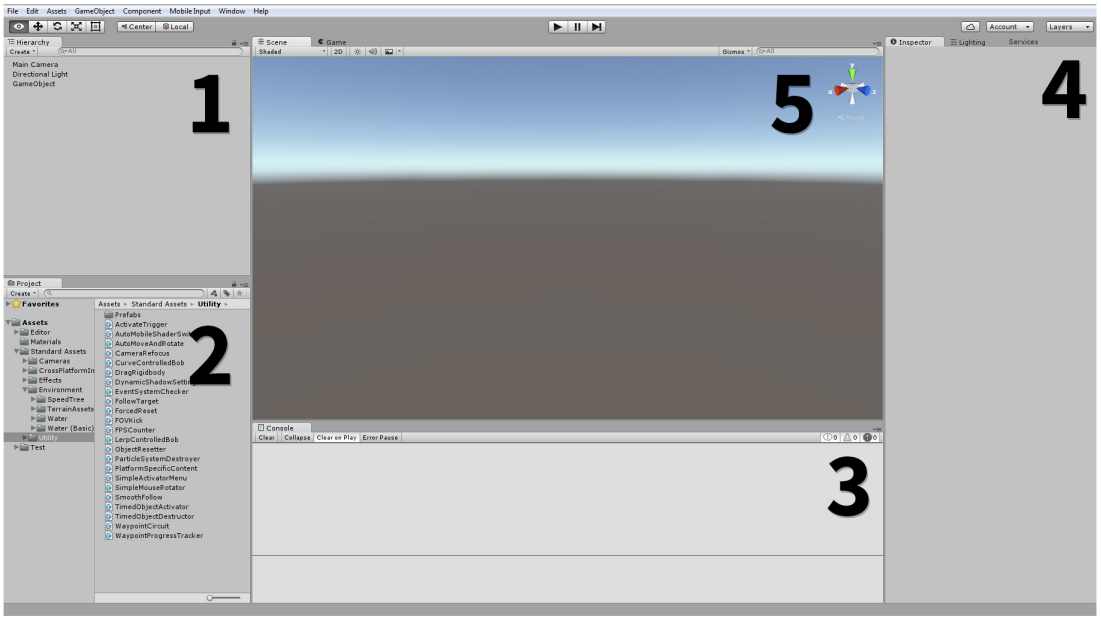

Figure 2.10: Unity editor, depicting the hierarchy window (1), project window (2), console (3), inspector (4), and scene view (5).

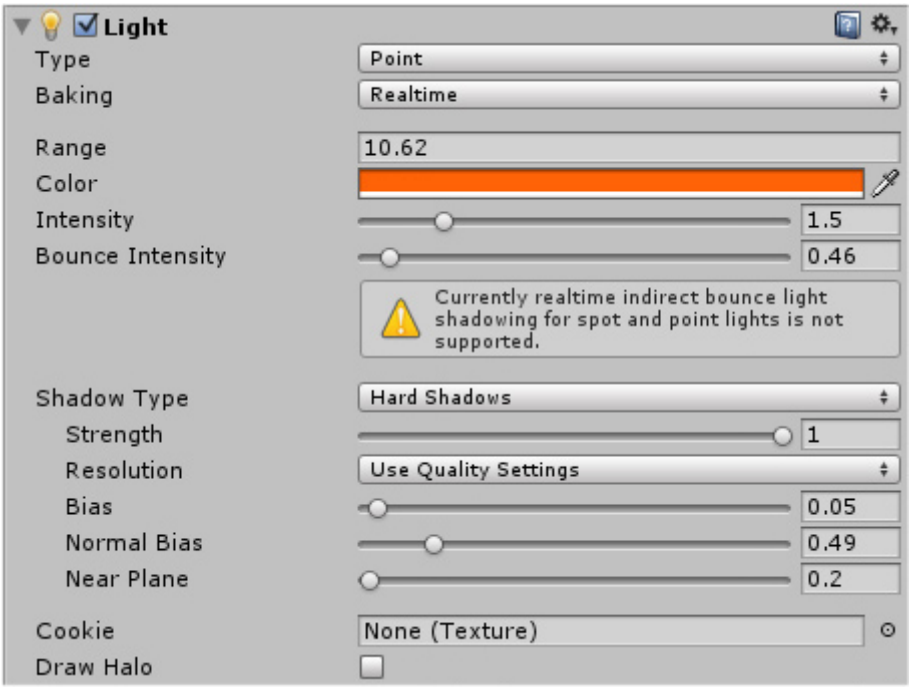

Figure 2.11: Inspector window, showing variables associated with a `Light` `Component`.

Inspector Window

The *inspector* is the Unity editor's interface for editing variables associated with `GameObjects` and `Components` [50], [53]. Unity's `EditorGUI` namespace contains functions which provide various controls and input methods for the various data members a `Component` can contain [54].

It is worth noting the diversity of input controls which can be specified by developers writing a *custom editor* - a script which defines user interface behaviour for a `Component` [55]. An example is provided in figure 2.11. The values set through these inputs map to properties within an instance of the `Component`.

2.3.6 Analytics Systems for Unity

Unity's "asset store" (introduced in section 2.3) features several sets of scripts (referred to as *plugins*) which provide analytics and telemetry collection capabilities [56]. At the time of writing, several analytics plugins were sold by their developers at costs ranging from $2 to $75 USD; however, plugins

for three commercial analytics services (GameAnalytics, Playtomic, and Kii Game Cloud) were available for free [56].

The free plugins were investigated for any 'modular' functionality. However, none was discovered. Each system is integrated in a similar manner to Unity's analytics service (described in section 2.3.7), requiring a method call which transmits data to a server. No evidence of a *component-based* approach to analytics collection was observed in any of the free plugins. Furthermore, each of the free plugins were developed with older versions of the Unity editor, and released prior to the introduction of Unity's analytics service; the Playtomic plugin was released in 2011, and the Kii and GameAnalytics plugins released in 2013 [56]. Updates to the Unity engine have rendered some functions used by these plugins obsolete, which caused problems when attempting to integrate them into an empty project. For instance, the errors shown in figure 2.12 were displayed when importing files from the GameAnalytics plugin.

Figure 2.12: Unity editor console, showing errors displayed upon import of GameAnalytics plugin

The paid plugins were not purchased or examined beyond the descriptions provided within the asset store interface. None of them appeared to adopt a *component-based* approach.

2.3.7 Unity Analytics Service

Unity's analytics service is relatively new, having been introduced in January 2015 [8]. It consists of a web-based interface for displaying collected telemetry data and generating metrics, a database system for storing the collected data, and a set of functions contained within the `UnityEngine.Analytics` namespace which are responsible for sending data to the analytics service.

The primary interface for the Unity analytics service is the *dashboard* - a webpage which displays and organizes telemetry data collected from a particular game. Unity Technologies provides a demo application and sample data to demonstrate the dashboard, which is depicted in figure 2.13.

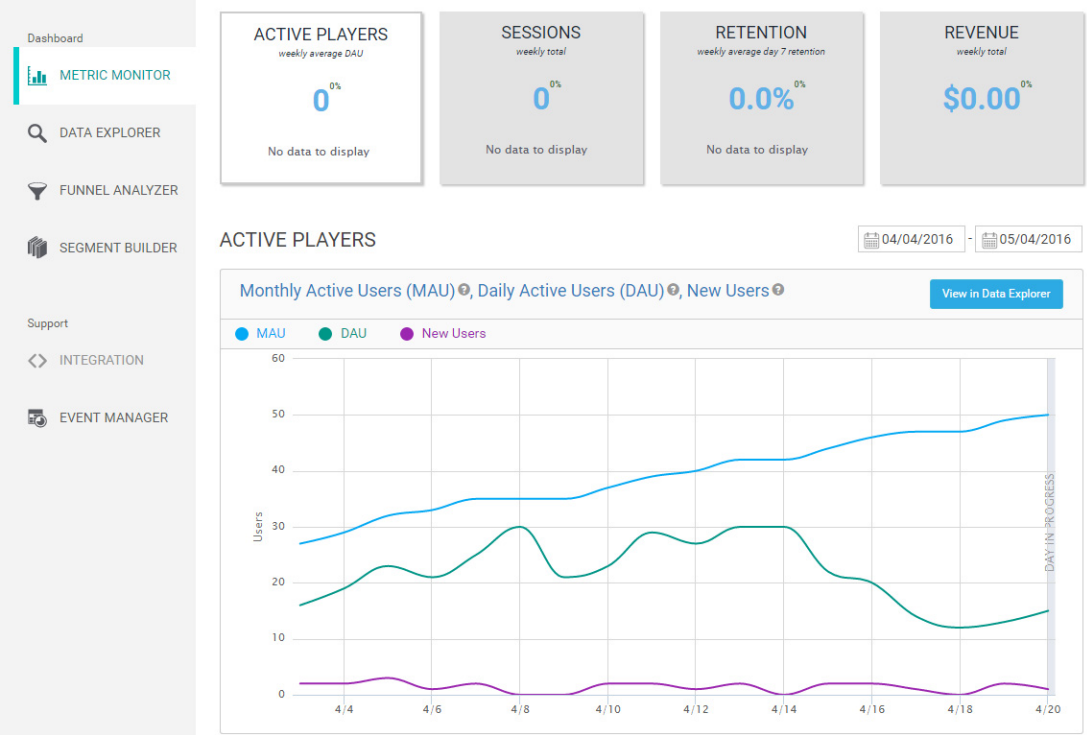

Figure 2.13: Unity analytics dashboard showing sample data collected from "Raging Bots" demo application

A public specification describing other components of the system (e.g., the data storage mechanism) has not been made available. However, some documentation exists which details functions available within the `Analytics` namespace [57], and limitations of the analytics service [58], [59]. A guide to integrating the analytics service into a Unity project is also provided [60].

Integration

Most of the steps for integrating Unity's analytics service into an existing project occur automatically. Developers generate a project identifier for their Unity game, and opt-in to the analytics service within the Unity editor [60]. This enables tracking of some basic user data, such as the number of unique daily users, or the user's device type [8], [58]. However, additional steps

must be taken if developers wish to track game-specific information. This is accomplished by the integration of *custom events* [58], [59].

Custom events are specified by programmers within a script file, by calling the `Analytics.CustomEvent` method [59]. The method signature is depicted in figure 2.14 - a `string` represents the name of the event, and an object implementing the `IDictionary` interface contains data related to the event. When the method is called, data contained in the `eventData` dictionary is transmitted to Unity's servers, and can be inspected in the dashboard. If the device runnning the application does not have an internet connection, the data is stored until a connection can be made [61].

```
1       Analytics.CustomEvent(string, IDictionary<string, object>);
```

Figure 2.14: `Analytics.CustomEvent` signature

The fact that custom events must be *hard-coded* into a script file make them somewhat inflexible by nature. If developers decide that the event should be triggered by another method, they must relocate the `CustomEvent` method call, and ensure that the dictionary object has appropriate access to any relevant parameters.

The `CustomEvent` method returns an enumerated type, an `Analytics.AnalyticsResult` [62]. This can be used by developers to identify any errors or problems which occur during the process of sending telemetry data to Unity's servers. It also highlights any limitations which developers may have encountered.

Limitations

The Unity analytics system enforces some limitations, particularly with regards to custom events. Some limitations are enforced within the `CustomEvent` method [59], while others appear to be enforced as a part of the web service [63].

If a limitation is encountered when the `CustomEvent` method is called, the method will return an `AnalyticsResult` describing the limitation [59]. These limitations are described in the Unity documentation as such:

- The custom event dictionary may contain a maximum of 10 parameters, or `AnalyticsResult.TooManyItems` is returned.
- Contents of the dictionary may not exceed 500 characters, or `AnalyticsResult.SizeLimitReached` is returned.
- Transmitting more than 100 custom events per hour causes `AnalyticsResult.TooManyRequests` to be returned.
- Attempting to transmit an event from an unsupported platform [8] will cause `AnalyticsResult.UnsupportedPlatform` to be returned.

The limitations with the web service are enforced by Unity Technologies. Each Unity project which uses the analytics service is allocated 1,000 “analysis points”, a unit representing a fixed amount of processing which Unity’s service applies to the analytics data [63]. Exceeding the allocated number of analysis points will disable data collection. Each custom event uses a certain amount of analysis points depending on its parameters [63]:

- A custom event without parameters costs one analysis point.
- Each numeric parameter in a custom event costs an additional point.
- The cost of non-numeric parameters is equal to the total number of *possible values* of the parameter. Boolean parameters cost two analysis points, while the cost of string parameters is variable.

These limitations have potential to create problems for games which interact with the Unity analytics service. Assuming a game was released to 500 players and has a string variable to track the player’s name, then there may be 500 individual values for the parameter - even more if players have a

chance to change their name. In this case, the custom event may cost over 500 analysis points - more than half of the points allocated to a project at the time of writing [63].

2.3.8 Survival Shooter Demo

Figure 2.15: Unity ‘Survival Shooter’ demo application

The ‘Survival Shooter’ demo is a Unity project distributed for free by Unity Technologies [64]. It provides example assets and scripts which comprise a simple game (depicted in figure 2.15). A corresponding tutorial details the how each component of the demo was developed and set up within the Unity editor [65].

Within the context of this project, the Survival Shooter demo was used as an evaluation platform. One of the project’s goals (see section 1.1) was to evaluate the developed components within an existing Unity application. The developed tools were incorporated into a copy of the Survival Shooter demo to gain an understanding of their practicality. This test application was also used in the performance evaluation process described in chapter 5.

2.4 Real-Time Performance

Games often run in real-time, so performance is essential. For instance, a game which renders 30 frames per second (FPS) only has 33 milliseconds to process each frame and display it to a player [66]. Therefore, any functions which can execute while a game is running must complete their execution in a matter of milliseconds, or execute *asynchronously* - decoupled from the main *game loop*.

Furthermore, requirements become more stringent when dealing with different devices, such as head-mounted virtual reality (VR) displays. Games designed for these devices must be particularly careful with regards to performance, since even small delays (upwards of 20 milliseconds) [67] can cause players to become disoriented, or aware of the delay [67], [68].

The components developed for this project do not specifically deal with rendering or gameplay. However, they must execute functions while a game is running. If a *component-based* analytics solution (described in sections 1 and 2.3.3) is to be effective in a gaming application, it is imperative that the components be computationally efficient.

Chapter 3

Design

The initial idea behind this project was a system which decouples programmers from Unity's analytics collection process. As described in section 2.3.7, programmers must specify the point at which data is to be collected, and populate a `Dictionary` object with data. This process is not friendly to users who lack programming knowledge.

The primary design objective is to provide non-programmers with an interface for manipulating the analytics collection process within a Unity project. This system should be *flexible* and capable of being used in a wide variety of games. A secondary objective is ensuring that the system remains *efficient* and capable of quickly processing analytics data.

3.1 Considerations

A primary design concern is how to interface with existing methods in order to transmit telemetry data. Unity's solution is the insertion of a method call at a specific point in a script where data is to be collected [59]. This is the same solution adopted by the third-party analytics plugins for Unity, as described in section 2.3.6.

Making a *flexible* solution ideally involves removing a hard-coded method call from a script file. Doing so, however, ultimately requires an *aspect-oriented* framework for C# (such as PostSharp [69]), which can *inject* code within an existing method. This is an unsuitable solution for this project, since developers would be forced to implement the aspect-oriented framework into their development process and project code.

Another key consideration is how designers and non-programmers inspect and specify data collection targets. A solution should ideally incorporate descriptive information about the collection point, and the data which is to be collected.

Also, as described in section 2.4, the real-time performance of any developed components is of concern. Component behaviours which take an inordinate amount of time to execute could cause delays in executing other game processes.

3.2 Development Approach

Dictating the development of this project's components is the manner in which telemetry data is to be captured at runtime. Given the inability to rely upon an aspect-oriented framework for this project (see section 3.1), the decision was made to rely on a hard-coded method call to capture the data. This is no different than the approach adopted by the existing Unity tools, as described in section 2.3.6 and 2.3.7.

Given this, the approach to development was modified to place a greater focus on *flexibility*. Although the initial design idea calls for removal of programmers from the process entirely, this was found to be impossible given the restrictions described in 3.1. As a result, the new goal became development of a system which enabled non-programmers to *quickly* change data collection targets, rather than the removal of programmers from the entire process.

A preliminary user study was not performed, since the project deals with the creation of an *interface* to Unity's existing analytics process. The Unity documentation explicitly defines the limitations provided by their analytics service (see section 2.3.7), and it was unclear how user input would be useful in advance of the development stage. However, a user study would be useful to gauge interest in the project, quantifying whether additional effort would be worthwhile. This is discussed further in chapter 7.

Development of the project's components was performed as an informal, *iterative* process, beginning with simple implementation of key features based on the design objectives (listed in section 3), and successively expanding upon them. In a sense, this is a form of *agile* development [49, pp. 3-4], although no specific agile methodology was employed. This was suitable given the size of the project team (a single developer), though a more rigid approach would have been necessary if other developers were involved.

3.3 Program Flow

Based on the design objectives provided in section 3, and the design considerations from section 3.1, a process, depicted in figure 3.1, was developed to describe the data collection procedure. This served as a rough guide for implementing the functionality of each class, and describes the behaviours implemented within this project.

The flowchart in figure 3.1 does not reflect certain prerequisite steps. Specifically, programmers must implement a method call into their code at the specific point where data is to be collected, and must mark this method with a custom attribute, `[AnalyticsEvent]`. Doing so 'enables' the method to work with this project's components. This process is discussed in section 4.2.3.

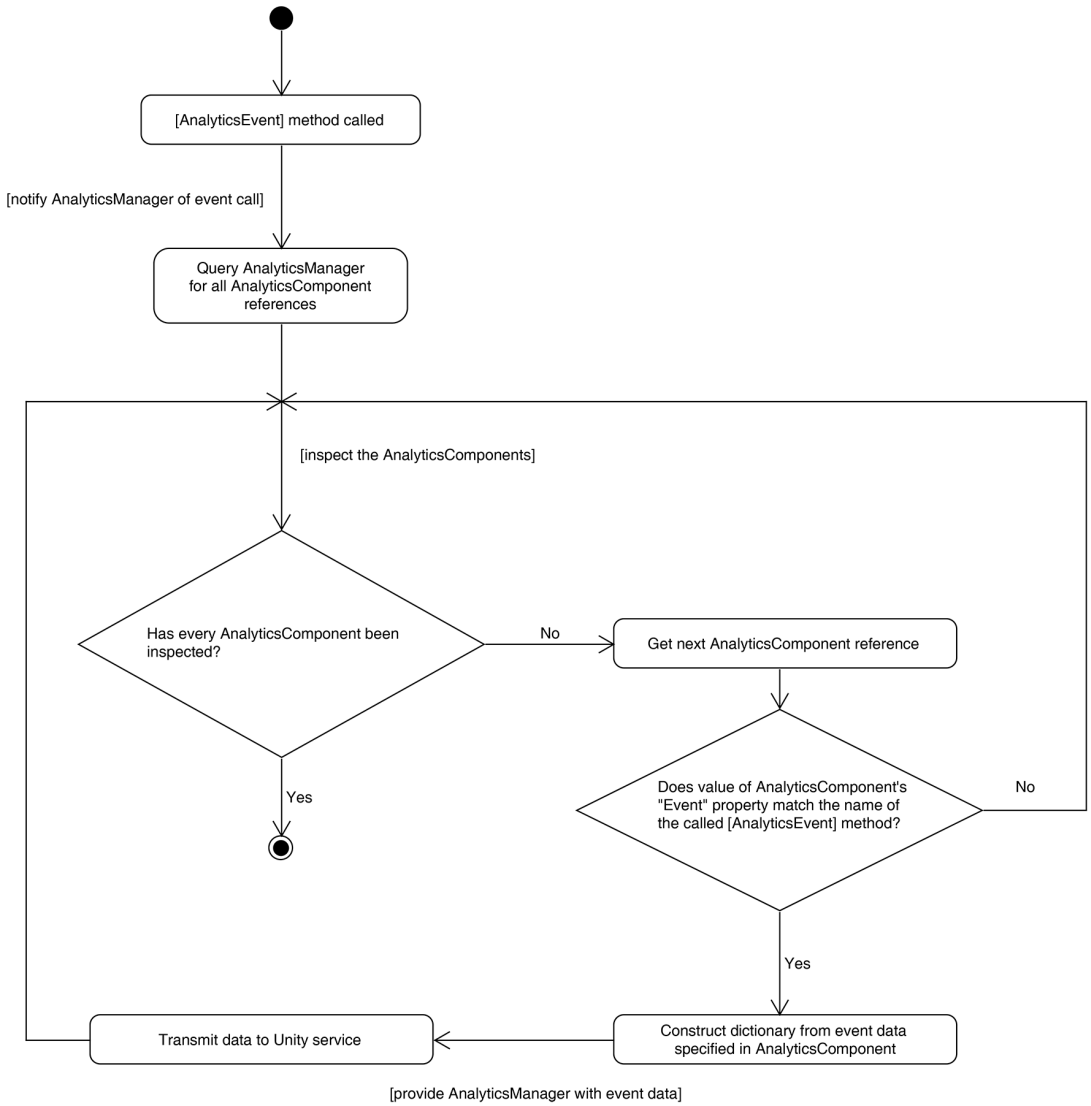

Figure 3.1: Data collection process diagram, updated to reflect final class names

Chapter 4

Classes and Implementation

The main functionality within this project lies within three core classes, described in section 4.1. Additional 'helper' classes and data structures are described in section 4.2. Describing the implementation of each class is intended to provide a holistic overview of the entire 'modular analytics' system which has been developed. A full guide to integration within a Unity project is provided on the DVD bundled with this report.

The user interface classes `AnalyticsSettingsWindow` and `AnalyticsComponentEditor` primarily consist of code which maps buttons and other user interface elements to underlying properties in the `AnalyticsSettings` and `AnalyticsComponent` classes. Since the focus is to provide an overview of the system's *functionality*, rather than its appearance, the interface scripts are only discussed where relevant.

Code in all classes was documented using the XML documentation style suggested by Microsoft [70]. Each script file was also reviewed with 'StyleCop' [71], a C# code analysis tool which identifies stylistic errors.

4.1 Core Classes

4.1.1 AnalyticsSettings Class

The `AnalyticsSettings` class handles *global* settings and behaviours for other analytics-related classes within the project. Its primary interface is provided through a window in the Unity editor application, shown in figure 4.1. The code which drives the user interface/apperance of the window is contained within a separate class - the `AnalyticsSettingsWindow`.

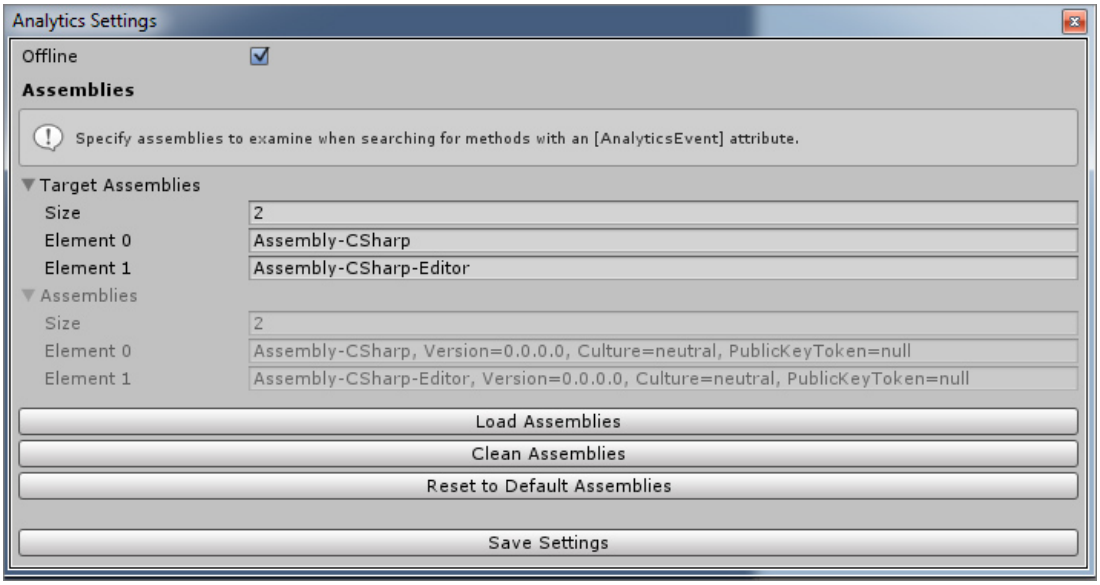

Figure 4.1: Analytics settings editor window

Displaying a free-floating editor window within the Unity editor requires an instance of the window object. As a result, the `AnalyticsSettings` class is implemented using a *singleton* design pattern, so that only a single instance may be created. An instance of the class is stored within a static '`Settings`' property, which can be accessed by other classes. The code for the '`Settings`' property is shown in appendix B.1.1. If an instance of the '`Settings`' object exists when the property is accessed, then the instance is returned - otherwise, initialization behaviour occurs and the '`Settings`' object is instantiated.

It is worth noting that the `AnalyticsSettings` class inherits from Unity's `ScriptableObject` class, rather than the `MonoBehaviour` class from which `Components` derive (see section 2.3.3). Deriving from `ScriptableObject` means that variables associated with the '`Settings`' object can be serialized. Because the '`Settings`' object exists independently of a Unity scene, the data is saved within an "asset" file within the Unity project.

Beyond storing *global* settings, the `AnalyticsSettings` class is responsible for loading assemblies containing code belonging to a Unity project. By default, two assemblies are loaded - `Assembly-CSharp` and `Assembly-CSharp-Editor` - which are built by the Unity editor. Functionality is provided in the `AnalyticsSettingsWindow` class to allow users to specify custom assemblies

(which were discussed in section 2.3). Loading assemblies is handled with the `LoadAssemblies` method, reproduced in appendix B.1.2.

Upon loading assemblies, the `AnalyticsSettings` class uses reflection to examine any methods contained within, identifying ones which have been marked with an `[AnalyticsEvent]` attribute (discussed in section 4.2.3). The method which provides this behaviour, `LoadMethods`, is reproduced in appendix B.1.3. The method names, and names of their containing `Types`, are stored in two properties belonging to the `AnalyticsSettings` class, 'MethodKeys' and 'MethodValues'. Values from these properties are used to determine the point at which telemetry data will be transmitted to the Unity service, a process discussed further in section 4.1.2.

Furthermore, an 'Offline' property belongs to the `AnalyticsSettings` class, and is displayed within the settings window. It provides a global option controlling whether data is sent to the Unity service. If the 'Offline' option is enabled, the data is *not* logged or stored locally, and simply displays in the editor application's console. This exists primarily as a 'debugging' option, and can be useful when attempting to perform a 'sanity check' of the analytics transmission procedure.

4.1.2 AnalyticsComponent Class

The `AnalyticsComponent` is a `Component` (see section 2.3.3) which can be attached to any `GameObject` within a Unity scene. An example is depicted in figure 4.2. The `AnalyticsComponent` provides a visual interface (controlled by the `AnalyticsComponentEditor` class) allowing users to manipulate the analytics collection process. Specifically, users specify a *trigger* method, and a series of *event keys* and *event values*. The event values are obtained from the *members* of `Components` attached to the `GameObject` (to which the `AnalyticsComponent` is also attached). This data is all contained within a property, 'EventData', which is a `List` of `AnalyticsData` objects (described in section 4.2.2).

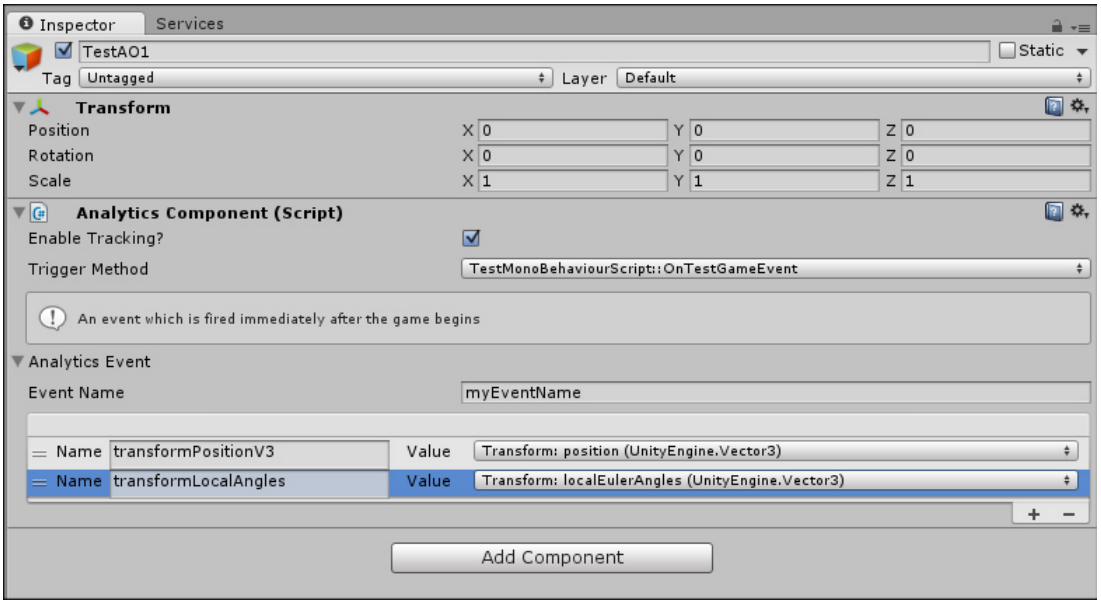

Figure 4.2: Analytics component, attached to a `GameObject`

This sounds complex, but the `AnalyticsComponent` serves a simple purpose - providing an interface for users to manipulate the collection process, and as a collection point for various *members* which can contribute telemetry data. Its functionality can be broken down into a series of steps:

- Retrieve a `MemberInfo` from the members of every `Component` attached to the `GameObject`(ignoring members of the `AnalyticsComponent`).

- Await user specification of an 'event', consisting of several components: the *trigger*, or data collection point, a user-specified *event name*, and the *event information* - a list of paired *event keys* and *event values* which will be sent as a `Dictionary` to the Unity service (see section 2.3.7).

- Wait until a call is made to this `AnalyticsComponent`'s `Send` method. When this call is received, construct a `Dictionary` from the event information.

- Call the `static` `AnalyticsManager`.`TransmitEvent` method, passing the *event name* and constructed `Dictionary` object as parameters.

MemberInfo Retrieval

The first step, retrieval of the `MemberInfo` objects, is handled by the `RetrieveMemberInfo` method, reproduced in appendix B.2.1. This method retrieves each `Component` attached to the `GameObject`, then uses the `System.Reflection` methods `GetProperties` and `GetFields` in the same manner as the example provided in section 2.2.6. Data is collected from each retrieved `MemberInfo`, and stored in an instance of an `AnalyticsMemberInfo`, so that the data can be serialized (see section 4.2.4). Some checking is performed against a `static` list within the `AnalyticsSettings` class, which contains certain `MemberInfo`s which will be ignored - primarily, these are obsolete properties created by Unity. The collected members are used to populate a dropdown list from which users select a specific member, as can be seen near the bottom of figure 4.2.

Retrieval of Nested Properties

One 'gotcha' with the `RetrieveMemberInfo` method (appendix B.2.1) is that any *nested* properties (properties which belong to a property) will not be retrieved. This is a design decision prioritizing convenience to the end-user. Many nested properties are not useful for analytical purposes - an example is the '`Chars`' property, which stores an array of `char` and belongs to the `string` class [38]. If nested properties were also retrieved by the `RetrieveMemberInfo` method, then every property of type `string` retrieved would also have a corresponding '`Chars`' property, inflating the list of collected members.

In some fringe use-cases, there are situations where accessing these nested properties may be valuable, such as accessing a specific component of a two- or three-component vector. Modifying the `RetrieveMemberInfo` method to do so is a trivial task - the code from figure 4.3 can be inserted in the loop which iterates the `PropertyInfo` array (see appendix B.2.1, line 18), and each nested property can be manipulated as necessary.

```
1  foreach PropertyInfo subProperty in property.PropertyType.GetProperties()
2  {
3      // do something with the subProperty
4  }
```

Figure 4.3: Code snippet demonstrating retrieval of nested properties

It is important to note that the code from figure 4.3 will only inspect one 'level' of nested properties. It is possible to recursively check properties until no further nested properties are discovered, although this may significantly lengthen the execution time of the `RetrieveMemberInfo` method.

Setting Trigger Methods

The 'trigger' method is specified by users by selecting an option from a dropdown list, as can be seen in figure 4.2. A small information box displays a short description which was specified by the `[AnalyticsEvent]` attribute (see section 4.2.3). Each entry in the dropdown list maps to a `AnalyticsTrigger` object, describing the method's name, and the `Type` with which it is associated. These two pieces of information are examined by the `AnalyticsManager` when any trigger event is fired, as described in section 4.1.3.

Specifying Event Data

Much like the 'trigger' method, the members which contribute to an event's data are also specified by the users, as can be seen near the bottom of figure 4.2. In this case, users may select up to 10 members (a limit set by Unity, see section 2.3.7) whose data they wish to send to the Unity service. Each member is paired with a user-specified name. The pairs of member and name data are stored within an `AnalyticsData` object, described in section 4.2.2.

Sending Event Data

The process of sending event data begins outside of the `AnalyticsComponent`. Much of this behaviour is handled by the `AnalyticsManager` (discussed in section 4.1.3).

When the Send method (reproduced in appendix B.2.2) executes, it attempts to construct a Dictionary<string, object> from the 'EventData' list, using the BuildEventDictionary method (reproduced in appendix B.2.3). This method uses the GetReflectedData method (appendix B.2.4), which uses reflection to capture values from members specified in the 'EventData' list. Once the data is bundled into a Dictionary, it is sent to the AnalyticsManager, which handles transmission of the data to the Unity service (see section 4.1.3).

4.1.3 AnalyticsManager Class

The static AnalyticsManager class serves as the centralized, 'point of contact' for the modular analytics system. It accepts inputs from methods marked with the [AnalyticsEvent]attribute, which serve as 'trigger' methods for the AnalyticsComponents. Specifically, when a method with the [AnalyticsEvent] attribute calls the AnalyticsManager.Notify method (reproduced in appendix B.3.1), the AnalyticsManager inspects every AnalyticsComponent, comparing the 'trigger' method with the calling method (whose name is sent as a parameter within the Notify method). The AnalyticsManager then calls the Send method on any AnalyticsComponents with matching trigger methods.

The AnalyticsComponent's Send method calls the AnalyticsManager's TransmitEvent method (appendix B.3.2), passing in a Dictionary of values relating to the analytics event. The TransmitEvent method contains a call which sends data to the Unity service, Analytics.CustomEvent, which was discussed in section 2.3.7. The AnalyticsManager handles any AnalyticsResponses sent by the Unity service, and displays them to the console them using the LogResult method (appendix B.3.3). Sample output from the LogResult method is shown in figure 4.4, demonstrating its use for debugging and local testing.

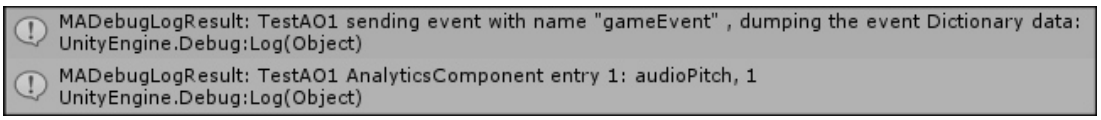

Figure 4.4: Output from LogResult method

4.2 Helper Classes

4.2.1 AnalyticsFunctions Class

The `AnalyticsFunctions` class contains `static`, general-purpose 'helper' methods used throughout the project. Several methods (`StripReturnType`, `StripNamespace`, `StripMethodParameters`, and `GetAssemblyShortName`) perform operations on string representations of `MemberInfo` and `MethodInfo` objects. The `AnalyticsComponent` also relies on a method contained within this class, `CleanPropertyString`, reproduced in appendix B.4.1. This method 'sanitizes' user-specified strings which are bundled with the `Dictionary` of telemetry data transmitted to the Unity service.

4.2.2 AnalyticsData Class

The `AnalyticsData` class provides a serializable structure for data items contained within `AnalyticsComponent` instances. Each instance of an `AnalyticsComponent` provides a list interface (see sections4.1.2 and figure 4.2) allowing users to specify which *members* from attached `Components` will have their values captured and sent to the Unity service. This list is comprised of `AnalyticsData` instances.

This class is comprised of three properties. The '`Index`' property represents the position of the data item within the `AnalyticsComponent`'s '`EventData`' list. The '`EventName`' property represents the name associated with the event data. Finally, the '`EventInfo`' property contains a `AnalyticsMemberInfo` instance, which provides information about the member whose data will be collected (see section 4.2.4).

The code for the `AnalyticsData` class has been provided in appendix B.5.

4.2.3 AnalyticsEvent Class

The `AnalyticsEvent` class defines a custom attribute, intended to be applied on methods in which a call to `AnalyticsManager.Notify` has been inserted.

The purpose of this is to identify any methods which can serve as a 'trigger' for sending data to the Unity analytics service, as discussed in sections 4.1.1, 4.1.3 and 4.2.3.

The `[AttributeUsage]` attribute has been applied to the `AnalyticsEvent` class, with a parameter of `AttributeTargets.Method`. This ensures that the `[AnalyticsEvent]` attribute may only be applied to methods [72], which is the intended behaviour.

The `[AnalyticsEvent]` attribute contains a string property, '`Description`'. This provides a space for a short description of the method, which is stored in the `AnalyticsSettings` instance (see section 4.1.1, and appendix B.1.4). This description is displayed to users within the `AnalyticsComponent` interface, and can be seen in figure 4.2.

The code for the `AnalyticsEvent` class has been provided in appendix B.6.

Applying AnalyticsEvent to Existing Code

To identify methods which can participate in the analytics collection process, a programmer must first mark them with the `[AnalyticsEvent]` attribute, and insert a method call to `AnalyticsManager.Notify` at the point where data should be sent. A code stub illustrating this is provided in figure 4.5.

```
1  [AnalyticsEvent("Description of OnFoo method")]
2  public void OnFoo()
3  {
4      AnalyticsManager.Notify("OnFoo", "TestMonoBehaviourScript");
5  }
```

Figure 4.5: Sample method demonstrating implementation of method call and attribute for modular analytics system

As discussed in sections 4.1.1 and 4.1.2, the `AnalyticsSettings` class collects a listing of every method with the `[AnalyticsEvent]` attribute, and uses this to populate the dropdown listing of 'trigger' methods in each `AnalyticsComponent`. String parameters provided to the `Notify` method enable identification of the specific method, and its containing type. The recommendation to programmers is to insert this method call and attribute

within *every* method which might be called during the execution of a Unity game; doing so means that users of the system have a greater selection of data collection events to choose from.

4.2.4 AnalyticsMemberInfo Class

The `AnalyticsMemberInfo` class provides a serializable representation of information retrieved from `MemberInfo` instances. It contains properties which represent a member's *reflected type* (the `Type` to which a member belongs), the member's own `Type`, its name, its `MemberType` [73], and its *assembly qualified name* [74] (containing the `Assembly` to which the member's `Type` belongs).

4.2.5 AnalyticsTrigger Class

The `AnalyticsTrigger` class serves as a serializable container for paired values extracted from a `MethodInfo` object. The class contains two properties, '`MethodValue`' and '`MethodKey`', and corresponding backing fields for these properties. The properties are populated based on a user's selection of a 'target method' within the `AnalyticsComponent`in the Unity editor. This selection allows indexing into properties belonging to the `AnalyticsSettings` object, which contain `MethodInfo` data retrieved from assemblies belonging to the Unity project.

This information is examined by the `AnalyticsManager` when determining which objects to collect telemetry data from. Specifically, it compares the 'triggering' method's class and method names against the values stored in the '`MethodValue`' and '`MethodKey`' properties.

Chapter 5

Evaluation

One of the goals described in section 1.1 is performance measurement - comparing the efficiency of the developed tools against the existing Unity analytics functions (that is, simply calling Unity's `Analytics.CustomEvent` method, described in section 2.3.7. Performance measurement is discussed in section 5.0.2. Prior to performance testing, a short test was undertaken to verify that values were being correctly transmitted to the Unity analytics service. This process is discussed in section 5.0.1.

5.0.1 Functionality Tests

Before performing any performance testing, it was important to verify that the system was fully operational. The system was evaluated periodically during development by using the 'Offline' option in the analytics settings panel (described in section 4.1.1). However, periodic testing involving transmission of data to the Unity service was *not* performed, primarily due to the event limitations imposed by the Unity service (see section 2.3.7).

To begin this process, two `GameObjects` were created, each with an `AnalyticsComponent` attached. Basic Unity `Components` were also attached - one object given a `Light` component, and the other an `AudioSource` component. Furthermore, a simple script (reproduced in appendix B.7) was created which included a method prepared for the analytics collection process (the preparation process was discussed in 4.2.3).

By using the interface of the `AnalyticsComponent`s, several members were selected for analysis. The state of the `AnalyticsComponent`, after setup, is depicted in figure 5.1. The members chosen *did not* cover the full range

of numeric types - more comprehensive testing is required in this regard. However, the purpose of this specific test was not a rigorous evaluation. Rather, it was intended as a 'sanity check', being a logical progression from local, offline testing performed throughout development.

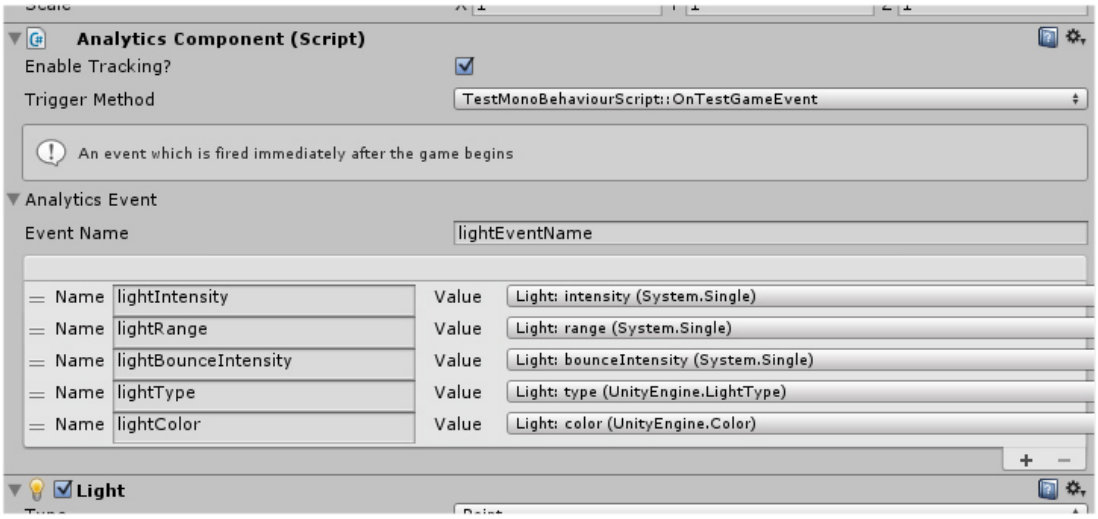

Figure 5.1: `AnalyticsComponent`, populated with values from `Light` component for testing

Once the `AnalyticsComponents` were prepared, the functionality was inspected in two ways. The first involves inspection of output from the `LogResult` method. The `AnalyticsManager` (see section 4.1.3) provides some output when attempting to send an event to the Unity service, but also provides the output of the `LogResult` method. This output includes an `AnalyticsResult` enumeration (discussed in section 2.3.7) describing whether the event was transmitted successfully. The output from this functionality testing is provided in figure 5.2. In this case, the Unity service returned an `AnalyticsResult.OK` enumeration, indicating that both events were received successfully.

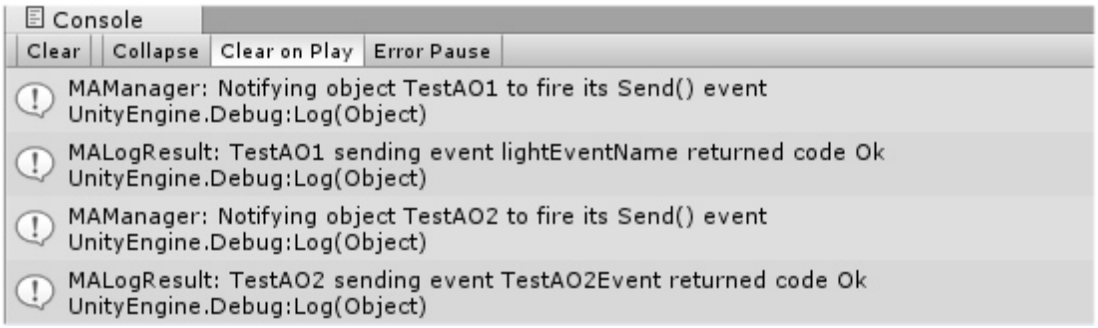

Figure 5.2: Output from `AnalyticsManager`, confirming event transmission, and demonstrating a response enumeration received from the Unity analytics service

It can take some time for the analytics dashboard (see section 2.3.7) to display data which has been received [75]. However, the 'Advanced Integration' tab of the Unity analytics system (figure 5.3) provides a small window in which received data will appear almost immediately, which can be used for debugging purposes. Inspecting the data displayed in this window (depicted in figure 5.4) confirms that the correct values were transmitted by the `AnalyticsComponent`. Since it has been demonstrated that the correct data was successfully transmitted to Unity's service, this test can be considered successful.

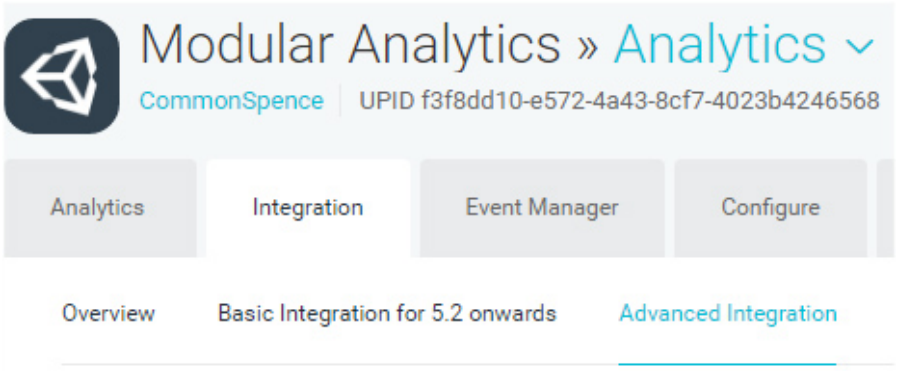

Figure 5.3: Unity analytics service, advanced integration tab

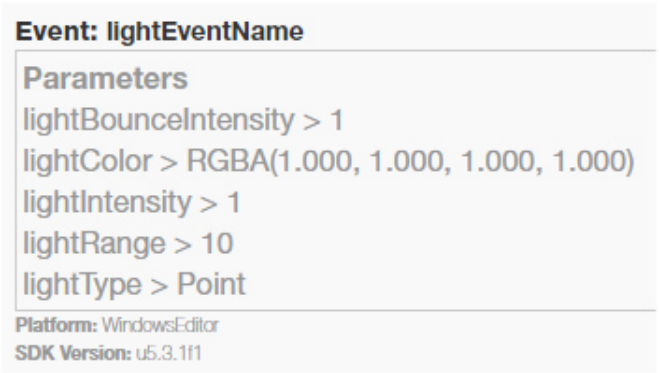

Figure 5.4: Inspecting values sent to Unity analytics service

5.0.2 Integration and Performance Tests

The importance of performance for real-time applications was discussed in section 2.4. Carmack and Abrash both note that frame rendering times exceeding 20 milliseconds can create noticeable delays for VR display devices [68], [67]. Since VR applications are very sensitive to frame rendering times, they provide a good metric for determining 'efficiency' of a software component designed for use in a real-time application. If the components

built for this project exceed 20ms to collect and transmit data, then it is a good indication that the developed solution may not provide an appropriate level of performance. However, depending on the results, the tools may be suitable for non-VR applications; Abrash notes that some non-VR games "have latency from mouse movement to screen update of 50ms or higher (sometimes much higher)" [67]. In this case, spending 5-10ms of a frame's rendering time for analytics collection may be appropriate, especially since analytics data does not need to be captured and transmitted every frame.

Beyond performance evaluation, another goal set for the project (refer to section 1.1) was the *integration* of the components into an existing Unity game. These goals were condensed, so that performance evaluation could occur within a 'practical' testing environment. The 'Survival Shooter' project, described in section 2.3.8, was duplicated, and the modular analytics scripts were imported.

The first step of integration was to modify the Survival Shooter scripts and define a `[AnalyticsEvent]` from which data could be collected. The '`PlayerHealth`' script belonging to the Survival Shooter project provides a method, '`Death`', which is called when the game ends. Following the procedure described in section 4.2.3, the '`Death`' method was modified to include the `[AnalyticsEvent]` attribute. This was a particularly convenient method for testing, since the game could be left running to collect performance data without any user interaction necessary.

A set of 'wrapper' functions was developed which encapsulate both the `Analytics.CustomEvent` and `AnalyticsManager.Notify` methods. Code for these functions is stored within the `static` `TimingFunctions` class, whose code is reproduced in appendix B.8. Within the wrapper functions, the `System.Diagnostics.Stopwatch` class is used to measure the execution time of each method in milliseconds. The times of each function were logged to tab-delimited text files.

It was decided that two tests would be performed - the first timing the transmission of a custom event with two parameters, and the second timing an event with six parameters. The results from these tests would indicate if further testing was required (e.g., to confirm if the time to transmit events scales based on the number of event parameters). The `AnalyticsComponent` was specified using members from the `Transform` component, as depicted in figures 5.5 and 5.6.

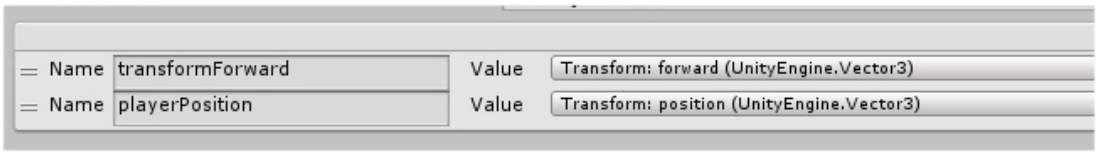

Figure 5.5: `AnalyticsComponent`, prepared for testing with two event parameters

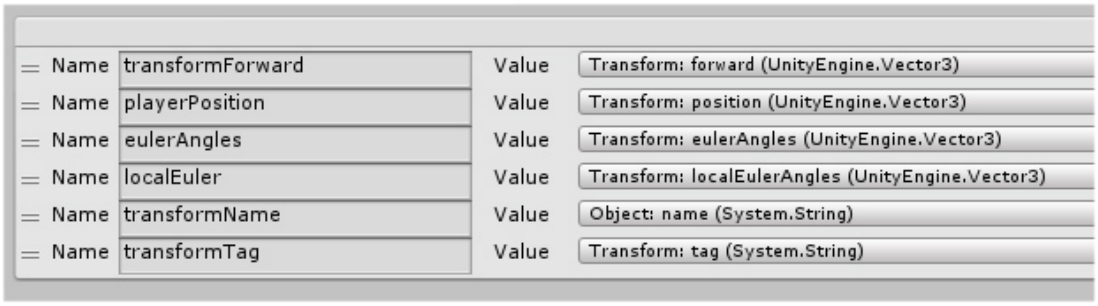

Figure 5.6: `AnalyticsComponent`, prepared for testing with six event parameters

Initially, 75 test cycles were performed. Each cycle transmitted two events - one from the `TimingFunctions.TimedCustomEvent` method, and another from the `TimingFunctions.TimedNotify` method. Test output was confirmed in two ways - the first, again, was console output from the `AnalyticsManager` and `TimingFunctions` classes. Owing to the number of events sent (see section 2.3.7), the `AnalyticsResult.TooManyRequests` enumeration was returned after a certain point. This was expected, and did not affect the timing procedures, but demonstrates that the system works appropriately even when too many events are transmitted.

The second means of evaluation is verification of data through the 'Advanced Integration' tab of the Unity analytics service, just like the functionality testing procedure in section 5.0.1. The data displayed here matched the data which was transmitted, confirming successful integration into an existing Unity project. As a result, the 'integration' portion of this test was deemed successful.

With regards to performance evaluation, the tab-delimited files written by the `TimingFunctions` were inspected. For each of the two tests, a '*timedNotify.csv*' and '*timedCE.csv*' file was written. Each entry in the files contained the execution time of either the `AnalyticsManager.Notify` or `Analytics.CustomEvent` methods. Graphs of the method execution times from both the two- and six-event tests are shown in figures 5.7 and 5.8, respectively.

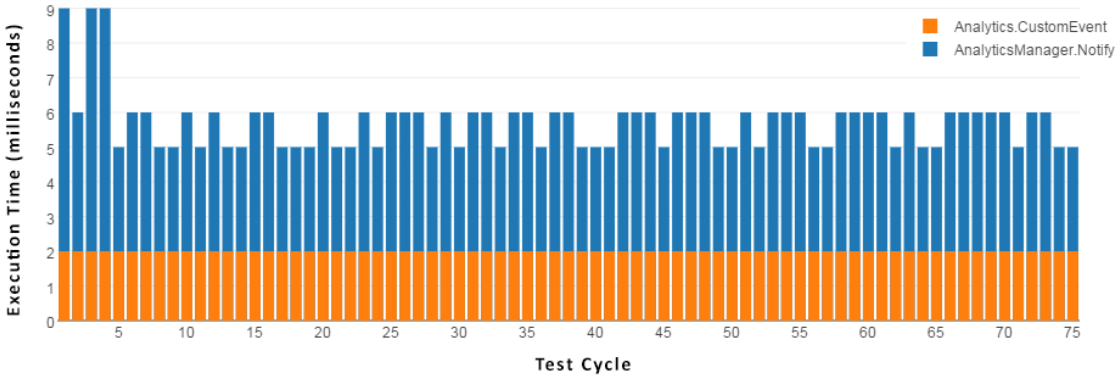

Figure 5.7: Method execution times from two-event test

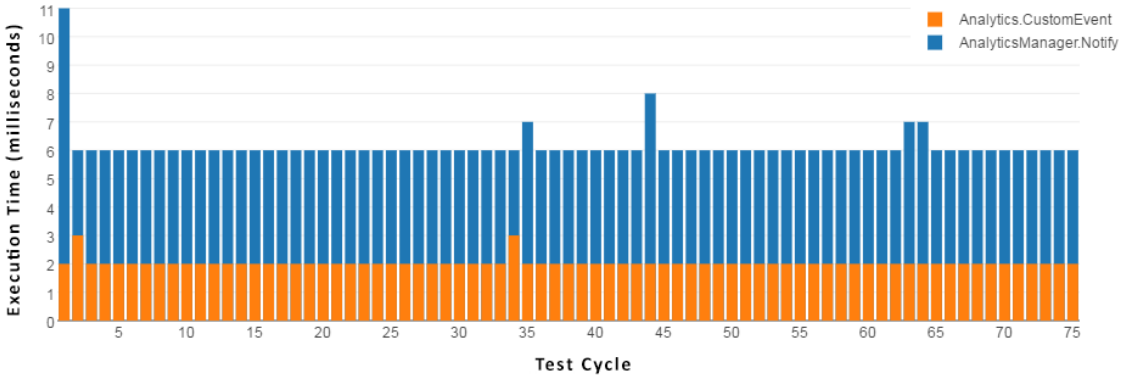

Figure 5.8: Method execution times from six-event test

The test results demonstrate that Unity's `Analytics.CustomEvent` method always outperformed the `AnalyticsManager.Notify` method. Since the `Notify` method includes a method call to `Analytics.CustomEvent` (as described in section 4.1.3), the `AnalyticsManager` can be seen as adding approximately 4 to 5 milliseconds to the telemetry collection procedure. This is not a tremendous amount of time, but it could have adverse effects on applications which are exceptionally sensitive to frame rendering times, such as games built for VR hardware.

It is also worth noting that the longest execution time of the `AnalyticsManager.Notify` method always occurred on its earliest calls. Since this method may be called infrequently, it is possible that the actual execution time experienced in an application may be closer to this value. A new test must be devised to determine if this is the case.

Given the additional functionality provided by the developed tools, a few additional milliseconds of processing seems like an appropriate trade-off. This seems particularly appropriate considering that communication must occur between the `AnalyticsComponent` and `AnalyticsManager`, and reflection must be used to retrieve the state of various members. It was deemed unnecessary to perform further tests, such as a test with ten parameters, since the execution time between the two- and six-parameter tests only increased by approximately 1 millisecond.

Chapter 6

Legal and Ethical Implications

Analytics systems, by nature, involve the collection of user data. Often, this data is not personally identifiable, so any legal or ethical issues relating to privacy may be more relaxed; European law specifically states that data protection principles “shall not apply to data rendered anonymous in such a way that the data subject is no longer identifiable” [76]. A key distinction for the tools developed as a part of this project is that they serve as an *interface* to Unity’s analytics service, rather than as a *stand-alone* system. All functions used to transmit and store data are a part of Unity’s `Analytics` namespace, as was described in section 2.3.7. Additionally, no storage mechanism to contain user data is provided as a part of these tools - all data is immediately transmitted to Unity’s service.

Unity Technologies provides a terms of service agreement for their analytics platform. It requires developers to provide a privacy policy stating how and why data is to be collected, and requires the consent of any parties whose data is to be collected, among other points [77]. This agreement places responsibility on developers using the service, so the author feels that there are few legal implications arising from the use of the tools developed for this project. Confirming this with legal counsel would be essential if these tools are to be distributed publicly, however. A separate agreement may be necessary, noting that developers who integrate these tools into their projects must also comply with Unity’s terms of service.

While there are social and ethical concerns about the collection and use of telemetry data [1, p. 30], such concerns are irrelevant in the context of this project - they would arise regardless of whether the tools developed in this project were employed in the data collection process.

Chapter 7

Conclusions and Future Work

In closing, a working prototype of a 'modular' analytics system has been developed. It demonstrates a functional integration with Unity's analytics service, as well as existing Unity applications, with code written by other developers. It also provides a graphical user interface to manage the *setup* of the analytics collection process, something that is not handled by any existing plugins for the Unity engine at the time of writing (see section 2.3.6). It is clear that the strong point of this set of tools is, at present, its ability to quickly specify variables which will contribute to an analytics collection system. There is also potential to use this system in an offline analytics collection process, as briefly touched upon in section 4.1.3.

There is still room for improvement in the tools developed. Many errors were identified and corrected during the development cycle, but some outstanding 'to-do' items are listed below:

- Handling of multiple `Components` of the same type attached to a single `GameObject`. At present, only the first `Component` can contribute its members to the data collection process.

- Customizable 'ignore' list for certain members, for de-cluttering the dropdown list of members in the `AnalyticsComponent`. Presently, a list of `MemberInfo`s which should be ignored is hard-coded into the `AnalyticsSettings` class.

- Validation of event data within the `BuildEventDictionary` method.

- Error-checking/redundancy within `AnalyticsFunctions` methods.

- ‘Hot-reloading’ of assemblies by the `AnalyticsSettings` class upon detecting changes to script files - currently, assemblies must be manually reloaded/reset by using the ‘Analytics Settings’ window.

- The ‘Analytics Settings’ window can, rarely, throw an exception which will cause the Unity editor to freeze. This has only been encountered when modifying the `AnalyticsSettingsWindow` script during development, but the exact cause of the exception should be identified.

- Usability improvements - describing, to users, sample values and applications of each available member (e.g., through tooltips in the `AnalyticsComponent` interface). Greater focus on making every aspect of the tools more friendly to non-programmers.

Some specific functionality must also be evaluated further - an example being the loading of custom assemblies. As mentioned in section 2.3, Unity can work with code written in many .NET-supported languages, as long as a .DLL file can be produced [30]. The `AnalyticsSettings` class provides an interface for specifying custom assemblies, but this feature has not been evaluated with an assembly built outside of the Unity editor. A multi-platform evaluation would also be necessary before releasing the tools, since Unity games support many different platforms [23]. Currently, the tools have only been evaluated on a Windows machine, so it is possible that unexpected behaviours may be encountered on different platforms.

Beyond this, rigorous testing is a necessary next step. Many potential use-cases have not been evaluated - such as multiple `AnalyticsComponents` with the same event names or values, whose data gets transmitted at the same time. It is expected that many bugs may be identified through usability testing in this manner, as different users can be expected to use the tools in different ways. A more generalized user study may prove useful, as well. If developers would be willing to sacrifice a small performance hit for a *flexible* set of analytics tools, then it may be worth improving upon the project further. As was mentioned above, no existing Unity plugins provide a user interface

for the analytics service, and therefore this set of tools may be a valuable addition to the Unity asset store. However, if many developers state that these tools would be unnecessary, then any further effort may be misguided, and better spent developing new tools which developers desire.

Despite shortcomings in the testing procedures, the system demonstrates that a ‘modular’ approach to the analytics collection process is indeed possible. The end result demonstrates achievement of the goals set in section 1.1, although there was insufficient time to perform a full usability study. Given further testing and iterative refinements, the developed tools may ultimately prove useful for Unity developers.

Appendix A

References

[1] M. S. El-Nasr et al., Eds., *Game Analytics: Maximizing the Value of Player Data*. Springer-Verlag London, 2013, ISBN: 9781447147695 (pp. 1, 3–6, 8, 46).

[2] P. Lankoski and S. Björk, *Game Research Methods*. ETC Press, 2015, ISBN: 9781312884731 (pp. 1, 4).

[3] K. Salen and S. Zimmerman, *Rules of Play: Game Design Fundamentals*. MIT Press, 2004, ISBN: 9780262240451 (p. 1).

[4] Association of Graduate Careers Advisory Services. (2016). Games developer job profile, [Online]. Available: `http://www.prospects.ac.uk/job-profiles/games-developer` (visited on 3rd Apr. 2016) (p. 1).

[5] J. Ward, Ed., *What is a Game Engine?*, GameCareerGuide.com, 29th Apr. 2008. [Online]. Available: `http://www.gamecareerguide.com/features/529/what_is_a_game_.php` (visited on 6th Apr. 2016) (pp. 1, 13).

[6] Unity Technologies. (n.d.). Unity - Fast Facts, [Online]. Available: `http://unity3d.com/public-relations` (visited on 5th Jan. 2016) (p. 1).

[7] Unity Technologies. (2014). Editor Version Release Dates, [Online]. Available: `http://web.archive.org/web/20140715224428/http://docs.unity3d.com/Manual/ReleaseDates.html` (visited on 11th Apr. 2016) (p. 1).

[8] S. Chan, Ed., *Announcing the Unity Analytics Open Beta*, The Unity Blog, 15th Jan. 2015. [Online]. Available: `http://blogs.unity3d.com/2015/01/15/announcing-the-unity-analytics-open-beta/` (visited on 27th Oct. 2015) (pp. 1, 19, 20, 22).

[9] W. Luton, *Free-to-Play: Making Money From Games You Give Away*. New Riders, 2013, ISBN: 9780321919014 (p. 5).

[10] J. Kim et al., 'Tracking Real-Time User Experience (TRUE): A comprehensive instrumentation solution for complex systems', presented at the CHI 2008, Florence, Italy, 2008, pp. 443–451. [Online]. Available: http://dl.acm.org/citation.cfm?id=1357126 (p. 6).

[11] Valve Software. (2014). Data to Drive Decision Making, Youtube, [Online]. Available: https://www.youtube.com/watch?v=HQwL6zh7AgA (visited on 6th May 2016) (p. 7).

[12] *Team Fortress 2*, Valve Software, 2007-2016. [Online]. Available: http://store.steampowered.com/app/440/ (visited on 10th May 2016) (p. 7).

[13] Valve Software, Ed., *Counter-Strike: Global Offensive » Heat Maps*, Counter-Strike Blog, 17th Apr. 2012. [Online]. Available: http://blog.counter-strike.net/index.php/2012/04/2704/ (visited on 9th May 2016) (p. 7).

[14] Valve Software. (2012). The Science of CS:GO - Heat Maps, [Online]. Available: http://blog.counter-strike.net/science/ (visited on 9th May 2016) (pp. 7, 8).

[15] Microsoft. (n.d.). Overview of the .NET Framework, [Online]. Available: https://msdn.microsoft.com/en-us/library/zw4w595w(v=vs.110).aspx (visited on 4th Feb. 2016) (p. 8).

[16] Microsoft. (n.d.). .NET Framework Class Library, [Online]. Available: https://msdn.microsoft.com/en-us/library/mt472912(v=vs.110).aspx (visited on 30th Jan. 2016) (p. 8).

[17] Microsoft. (n.d.). Using Namespaces (C# Programming Guide), [Online]. Available: https://msdn.microsoft.com/en-us/library/dfb3cx8s.aspx (visited on 9th Mar. 2016) (p. 8).

[18] Microsoft. (n.d.). What Is Managed Code?, [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/bb318664(v=vs.85).aspx (visited on 16th Feb. 2016) (p. 8).

[19] J. Richter, *CLR via C#*. Microsoft Press, 2012, ISBN: 9780735667457 (pp. 8, 9, 11).

[20] T. Thai and H. Lam, *.NET Framework Essentials*. O'Reilly Media, 2003, ISBN: 9780596005054 (pp. 8, 9).

[21] Microsoft. (n.d.). Common Language Runtime (CLR), [Online]. Available: https://msdn.microsoft.com/en-us/library/8bs2ecf4 (visited on 30th Jan. 2016) (p. 8).

[22] .NET Foundation. (n.d.). The Mono Runtime, [Online]. Available: http://www.mono-project.com/docs/advanced/runtime/ (visited on 19th Feb. 2016) (p. 9).

[23] Unity Technologies. (n.d.). Unity - Multiplatform - Publish your game to over 10 platforms, [Online]. Available: https://unity3d.com/unity/multiplatform (visited on 2nd Apr. 2016) (pp. 9, 48).

[24] R. Hauwert, Ed., *The Future of Scripting in Unity*, The Unity Blog, 20th May 2014. [Online]. Available: http://blogs.unity3d.com/2014/05/20/the-future-of-scripting-in-unity/ (visited on 19th Feb. 2016) (p. 9).

[25] Microsoft. (n.d.). Metadata and Self-Describing Components, [Online]. Available: https://msdn.microsoft.com/en-us/library/xcd8txaw(v=vs.110).aspx (visited on 10th Mar. 2016) (p. 9).

[26] Microsoft. (n.d.). Extending Metadata Using Attributes, [Online]. Available: https://msdn.microsoft.com/en-us/library/5x6cd29c(v=vs.110).aspx (visited on 10th Jan. 2016) (p. 9).

[27] Microsoft. (n.d.). Attribute Class, [Online]. Available: https://msdn.microsoft.com/en-us/library/system.attribute(v=vs.110).aspx (visited on 12th Mar. 2016) (p. 9).

[28] Microsoft. (n.d.). C#, [Online]. Available: https://msdn.microsoft.com/en-us/library/kx37x362.aspx (visited on 1st Mar. 2016) (p. 10).

[29] Unity Technologies. (n.d.). Unity - Manual: Creating and Using Scripts, [Online]. Available: http://docs.unity3d.com/Manual/CreatingAndUsingScripts.html (visited on 17th Dec. 2015) (pp. 10, 13).

[30] Unity Technologies. (n.d.). Unity - Manual: Managed Plugins, [Online]. Available: http://docs.unity3d.com/Manual/UsingDLL.html (visited on 7th Dec. 2015) (pp. 10, 13, 48).

[31] Microsoft. (n.d.). Types (C# Programming Guide), [Online]. Available: https://msdn.microsoft.com/en-us/library/ms173104.aspx (visited on 9th Mar. 2016) (p. 10).

[32] Microsoft. (n.d.). Members (C# Programming Guide), [Online]. Available: https://msdn.microsoft.com/en-us/library/ms173113.aspx (visited on 9th Mar. 2016) (p. 10).

[33] Microsoft. (n.d.). Properties (C# Programming Guide), [Online]. Available: https://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx (visited on 9th Mar. 2016) (p. 10).

[34] Microsoft. (n.d.). Basic Serialization, [Online]. Available: https://msdn.microsoft.com/en-us/library/4abbf6k0(v=vs.110).aspx (visited on 4th Jan. 2016) (p. 11).

[35] Microsoft. (n.d.). Reflection in the .NET Framework, [Online]. Available: https://msdn.microsoft.com/en-us/library/f7ykdhsy(v=vs.110).aspx (visited on 4th Jan. 2016) (p. 11).

[36] Microsoft. (n.d.). MemberInfo Class (System.Reflection), [Online]. Available: https://msdn.microsoft.com/en-us/library/system.reflection.memberinfo(v=vs.110).aspx (visited on 10th Mar. 2016) (p. 11).

[37] Microsoft. (n.d.). object (C# Reference), [Online]. Available: https://msdn.microsoft.com/en-GB/library/9kkx3h3c.aspx (visited on 9th May 2016) (p. 12).

[38] Microsoft. (n.d.). String Class (System), [Online]. Available: https://msdn.microsoft.com/en-us/library/system.string(v=vs.110).aspx (visited on 10th May 2016) (pp. 12, 33).

[39] Unity Technologies. (n.d.). Unity - Editor, [Online]. Available: https://unity3d.com/unity/editor (visited on 6th Apr. 2016) (p. 13).

[40] Unity Technologies. (n.d.). Asset Store, [Online]. Available: https://www.assetstore.unity3d.com/ (visited on 20th Dec. 2015) (p. 13).

[41] Unity Technologies. (n.d.). Unity - Manual: GameObjects, [Online]. Available: http://docs.unity3d.com/Manual/GameObjects.html (visited on 16th Dec. 2015) (pp. 13, 14).

[42] Unity Technologies. (n.d.). Unity - Manual: Using Components, [Online]. Available: http://docs.unity3d.com/Manual/UsingComponents.html (visited on 16th Dec. 2015) (pp. 13, 15).

[43] Unity Technologies. (n.d.). Unity - Scripting API: MonoBehaviour, [Online]. Available: http://docs.unity3d.com/ScriptReference/MonoBehaviour.html (visited on 8th May 2016) (p. 13).

[44] Unity Technologies. (n.d.). Unity - Scripting API: Object.Instantiate, [Online]. Available: http://docs.unity3d.com/ScriptReference/Object.Instantiate.html (visited on 11th May 2016) (p. 14).

[45] Unity Technologies. (n.d.). Unity - Manual: GameObject, [Online]. Available: http://docs.unity3d.com/Manual/class-GameObject.html (visited on 11th May 2016) (p. 14).

[46] Unity Technologies. (n.d.). Unity - Scripting API: Component, [Online]. Available: http://docs.unity3d.com/ScriptReference/Component.html (visited on 11th May 2016) (p. 14).

[47] Unity Technologies. (n.d.). Unity - Scripting API: GameObject, [Online]. Available: http://docs.unity3d.com/ScriptReference/GameObject.html (visited on 11th May 2016) (p. 14).

[48] Unity Technologies. (n.d.). Unity - Scripting API: Transform, [Online]. Available: http://docs.unity3d.com/ScriptReference/Transform.html (visited on 11th May 2016) (p. 15).

[49] G. M. Hall, *Adaptive Code via C#*. Microsoft Press, 2014, ISBN: 9780735683204 (pp. 15, 27).

[50] Unity Technologies. (n.d.). Unity - Manual: Variables and the Inspector, [Online]. Available: http://docs.unity3d.com/Manual/VariablesAndTheInspector.html (visited on 12th May 2016) (pp. 16, 18).

[51] Unity Technologies. (n.d.). Unity - Manual: Script Serialization, [Online]. Available: http://docs.unity3d.com/Manual/script-Serialization.html (visited on 5th Jan. 2016) (p. 16).

[52] Unity Technologies. (n.d.). Unity - Scripting API: SerializeField, [Online]. Available: http://docs.unity3d.com/ScriptReference/SerializeField.html (visited on 5th Jan. 2016) (pp. 16, 17).

[53] Unity Technologies. (n.d.). Unity - Manual: The Inspector Window, [Online]. Available: http://docs.unity3d.com/Manual/UsingTheInspector.html (visited on 12th May 2016) (p. 18).

[54] Unity Technologies. (n.d.). Unity - Scripting API: EditorGUI, [Online]. Available: http://docs.unity3d.com/ScriptReference/EditorGUI.html (visited on 12th May 2016) (p. 18).

[55] Unity Technologies. (n.d.). Unity - Manual: Custom Editors, [Online]. Available: http://docs.unity3d.com/Manual/editor-CustomEditors.html (visited on 16th Jan. 2016) (p. 18).

[56] Unity Technologies. (n.d.). Services/Analytics - Asset Store, [Online]. Available: https://www.assetstore.unity3d.com/en/#!/search/page=1/sortby=popularity/query=category:132 (visited on 20th Dec. 2015) (pp. 18, 19).

[57] Unity Technologies. (n.d.). Unity - Scripting API: Analytics, [Online]. Available: http://docs.unity3d.com/ScriptReference/Analytics.Analytics.html (visited on 8th Feb. 2016) (p. 20).

[58] Unity Technologies. (n.d.). Unity - Intro to Custom Events, [Online]. Available: http://unity3d.com/learn/tutorials/analytics/intro-to-custom-events (visited on 9th Feb. 2016) (pp. 20, 21).

[59] Unity Technologies. (n.d.). Unity - Manual: Custom Events, [Online]. Available: http://docs.unity3d.com/Manual/UnityAnalyticsCustomEvents.html (visited on 8th Feb. 2016) (pp. 20–22, 25).

[60] Unity Technologies. (n.d.). Unity - Manual: Setting Up Analytics, [Online]. Available: http://docs.unity3d.com/Manual/UnityAnalyticsOverview.html (visited on 9th Feb. 2016) (p. 20).

[61] M. Tanenbaum and Unity Technologies, Eds., *New Custom Event Limits - Please Read*, Unity3D Forums, 22nd Nov. 2015. [Online]. Available: http://forum.unity3d.com/threads/is-unity-analytics-a-best-effort-delivery-service-offline-unreliable-connections.311369/ (visited on 1st May 2016) (p. 21).

[62] Unity Technologies. (n.d.). Unity - Scripting API: Analytics.CustomEvent, [Online]. Available: http://docs.unity3d.com/ScriptReference/Analytics.Analytics.CustomEvent.html (visited on 9th Feb. 2016) (p. 21).

[63] A. Ferro and Unity Technologies, Eds., *New Custom Event Limits - Please Read*, Unity3D Forums, 1st Apr. 2015. [Online]. Available: http://

forum.unity3d.com/threads/new-custom-event-limits-please-read.315594/ (visited on 20th Jan. 2016) (pp. 21–23).

[64] Unity Technologies. (2015). Survival Shooter - Asset Store, [Online]. Available: https://www.assetstore.unity3d.com/en/#!/content/40756 (visited on 24th Nov. 2015) (p. 23).

[65] Unity Technologies. (n.d.). Unity - Survival Shooter tutorial, [Online]. Available: https://unity3d.com/learn/tutorials/projects/survival-shooter-project (visited on 2nd Dec. 2015) (p. 23).

[66] S. Wasson, Ed., *Inside the second: A new look at game benchmarking*, The Tech Report, 8th Sep. 2011. [Online]. Available: http://techreport.com/review/21516/inside-the-second-a-new-look-at-game-benchmarking (visited on 4th Apr. 2016) (p. 24).

[67] M. Abrash, Ed., *Latency - the* sine qua non *of AR and VR*, Ramblings in Valve Time, 29th Dec. 2013. [Online]. Available: http://blogs.valvesoftware.com/abrash/latency-the-sine-qua-non-of-ar-and-vr/ (visited on 3rd Apr. 2016) (pp. 24, 41, 42).

[68] J. Carmack, Ed., *Latency Mitigation Strategies*, #AltDevBlog, 19th Jul. 2014. [Online]. Available: https://web.archive.org/web/20140719085135/http://www.altdev.co/2013/02/22/latency-mitigation-strategies/ (visited on 3rd Apr. 2016) (pp. 24, 41).

[69] SharpCrafters s.r.o. (n.d.). PostSharp - The #1 pattern-aware extension to C# and VB, [Online]. Available: https://www.postsharp.net/ (visited on 16th Jan. 2016) (p. 25).

[70] Microsoft. (n.d.). XML Documentation Comments (C# Programming Guide), [Online]. Available: https://msdn.microsoft.com/en-us/library/b2s063f7(v=vs.100).aspx (visited on 13th May 2016) (p. 29).

[71] *StyleCop*, Microsoft, 2010-2016. [Online]. Available: https://stylecop.codeplex.com/ (visited on 29th Mar. 2016) (p. 29).

[72] Microsoft. (n.d.). AttributeTargets Enumeration (System), [Online]. Available: https://msdn.microsoft.com/en-us/library/system.attributetargets(v=vs.110).aspx (visited on 18th Feb. 2016) (p. 37).

[73] Microsoft. (n.d.). MemberInfo.MemberType Property (System.Reflection), [Online]. Available: https://msdn.microsoft.

com/en-us/library/system.reflection.memberinfo.membertype(v=vs.110).aspx (visited on 10th Mar. 2016) (p. 38).

[74] Microsoft. (n.d.). Type.AssemblyQualifiedName Property (System), [Online]. Available: https://msdn.microsoft.com/en-us/library/system.type.assemblyqualifiedname(v=vs.110).aspx (visited on 21st Feb. 2016) (p. 38).

[75] S. Chan and Unity Technologies, Eds., *Unity Analytics FAQs*, Unity3D Forums, 21st Jan. 2015. [Online]. Available: http://forum.unity3d.com/threads/unity-analytics-faqs.292372/ (visited on 22nd Apr. 2016) (p. 41).

[76] European Commission, 'Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data', *Official Journal*, pp. L281/31, 1995. [Online]. Available: http://ec.europa.eu/justice/policies/privacy/docs/95-46-ce/dir1995-46_part1_en.pdf (p. 46).

[77] Unity Technologies. (2015). Unity Analytics Terms of Service, [Online]. Available: https://analytics.cloud.unity3d.com/terms_of_service (visited on 30th Oct. 2015) (p. 46).

Appendix B

Selected Code

Selected code fragments are reproduced below. To provide more concise examples, some comments have been removed. An attached DVD contains the full source code, an integration guide, and a Unity asset package (which can be imported into new projects).

B.1 AnalyticsSettings.cs

B.1.1 Settings Property

```
1  public static AnalyticsSettings Settings
2  {
3      get // look for a settings file. if there isn't one, create it and assign it to settings
4      {   // also, populate the targetassemblies List from the list of defaults
5          if (AssetDatabase.LoadAssetAtPath(AssetPath, typeof(AnalyticsSettings)) == null)
6          {
7              // this is a ScriptableObject, so use CreateInstance<T> instead of the ctor
8              settings = CreateInstance<AnalyticsSettings>();
9
10             try
11             {
12                 // create the settings folder if it doesn't exist in the project environment
13                 if (!AssetDatabase.IsValidFolder("Assets/Editor/MATools/Settings"))
14                 {
15                     Debug.Log("MASettings: Creating settings folder at Assets/Editor/MATools/
                           Settings/");
16                     AssetDatabase.CreateFolder("Assets/Editor/MATools", "Settings");
17                 }
18
19                 // build the settings .asset file if it didn't exist already
20                 AssetDatabase.CreateAsset(settings, AssetPath);
21             }
22             catch (Exception e)
23             {
24                 if (e is NullReferenceException)
25                 {
```

```
                    Debug.LogError("MASettings: Loading the Analytics Settings threw a
                        NullReferenceException. Close and restart the Unity Editor.");
                }
            }

            settings.ResetDefaultAssemblies();
        }
        else
        {   // return the settings from the loaded .asset file
            settings = (AnalyticsSettings)AssetDatabase.LoadAssetAtPath(
                AssetPath, typeof(AnalyticsSettings));
        }
        return settings;
    }

    set
    {
        settings = value;
    }
}
```

B.1.2 LoadAssemblies Method

```
public void LoadAssemblies()
{
    foreach (string s in Settings.targetAssemblies)
    {
        try
        {   // test loading the Assembly
            Assembly a = Assembly.Load(s);
            if (!assemblies.Contains(a.ToString()))
            {
                assemblies.Add(a.ToString());
                Debug.Log("MASettings: Loading assembly " + a.ToString());
            }
            else
            {   // we won't load this because the string was a duplicate
                Debug.LogWarning("MASettings: Skipping load of assembly \"" + AnalyticsFunctions.
                     GetAssemblyShortName(a.FullName.ToString())
                    + "\" - the assembly was already loaded.");
            }
        }
        catch (Exception e) // the assembly was bad - don't load + warn the user
        {
            if (e is FileNotFoundException ||
                e is IOException || e is ArgumentNullException)
            {
                Debug.LogError("MASettings: The specified assembly string \""
                    + s + "\" is null, or could not be loaded.");
            }
```

```
            else if (e is BadImageFormatException)
            {
                Debug.LogError("MASettings: The specified assembly string \""
                    + s + "\" is not a valid assembly.");
            }
        }
    }

    LoadMethods();
}
```

B.1.3 LoadMethods Method

```
public void LoadMethods()
{
    methodKeys.Clear(); // clear existing methodinfo k/vs
    methodValues.Clear();
    foreach (string a in Settings.Assemblies)
    {
        Assembly ay = Assembly.Load(a);

        foreach (Type t in ay.GetTypes()) // check each Type in the assembly
        {
            MethodInfo[] methodInfos = t.GetMethods(BindingFlags.NonPublic | BindingFlags.Instance |
                  BindingFlags.Public | BindingFlags.Static); // get all Methods from the current
                Type

            foreach (MethodInfo m in methodInfos) // check each Method for an AnalyticsEvent
                 attribute
            {
                if (m.GetCustomAttributes(typeof(AnalyticsEvent), true).Length > 0)
                {
                    string mK = AnalyticsFunctions.StripReturnType(m.ToString())[1];
                    mK = AnalyticsFunctions.StripMethodParameters(mK)[0];

                    methodKeys.Add(mK); // stores the method which had attribute [AnalyticsEvent]
                    methodValues.Add(m.ReflectedType.ToString()); // stores the Type containing the
                         method above
                }
            }
        }
    }

    LoadMethodDescs();
}
```

B.1.4 LoadMethodDescs Method

```
1  public void LoadMethodDescs()
2  {
3      methodDescs.Clear();
4
5      int i = 0;
6      foreach (string s in MethodKeys)
7      {
8          Type t = Type.GetType(MethodValues[i]);
9          AnalyticsEvent e = (AnalyticsEvent)t.GetMethod(s, BindingFlags.NonPublic | BindingFlags.
               Instance | BindingFlags.Public | BindingFlags.Static).GetCustomAttributes(typeof(
               AnalyticsEvent), true)[0];
10         MethodDescs.Add(e.Description);
11         i++;
12     }
13 }
```

B.2 AnalyticsComponent.cs

B.2.1 RetrieveMemberInfo Method

```
1  public void RetrieveMemberInfo()
2  {
3      // clear out component lists (to avoid old component data being present)
4      memberInfos.Clear();
5      stringMemberInfos.Clear();
6
7      // check each Component attached to the gameObject container
8      foreach (Component c in gameObject.GetComponents(typeof(Component)))
9      {
10         Type t = c.GetType(); // get the component's Type
11
12         if (t != typeof(AnalyticsComponent))
13         {
14
15             PropertyInfo[] props = t.GetProperties(); // get all public properties
16             FieldInfo[] fields = t.GetFields();       // get all public fields
17
18             foreach (PropertyInfo p in props)   // check each property
19             {
20                 if (!MemberInfoIgnored(p.ToString())) // check property string against global
                       ignores
21                 {
22                     memberInfos.Add(new AnalyticsMemberInfo(
23                         p.ReflectedType.ToString(), p.PropertyType.ToString(), p.Name.ToString(), p.
                               MemberType, p.ReflectedType.AssemblyQualifiedName));
24                 }
25             }
26
27             foreach (FieldInfo f in fields) // check each field
28             {
29                 if (!MemberInfoIgnored(f.ToString())) // check field string against global ignores
30                 {
31                     memberInfos.Add(new AnalyticsMemberInfo(
32                         f.ReflectedType.ToString(), f.FieldType.ToString(), f.Name.ToString(), f.
                               MemberType, f.ReflectedType.AssemblyQualifiedName));
33                 }
34             }
35         }
36     }
37
38     FormatMemberInfoStrings(); // build the MemberInfo strings for the CustomEditor Popup
39 }
```

B.2.2 Send Method

```
1  public void Send()
2  {
3      if (trackingEnabled)
4      {
5          Dictionary<string, object> d = BuildEventDictionary();
6          AnalyticsManager.TransmitEvent(eventName, d, gameObject);
7      }
8      else
9      {   // I'm !trackingEnabled....don't send my information, but let the user know why
10         Debug.LogWarning("MAComponent: Analytics Component attached to " + name + " was instructed
               to Send(), but has tracking disabled.");
11     }
12 }
```

B.2.3 BuildEventDictionary Method

```
1  public Dictionary<string, object> BuildEventDictionary()
2  {
3      Dictionary<string, object> eventDataDict = new Dictionary<string, object>();
4
5      // NOTE: A populated dictionary is NOT required by the Unity service. If this behaviour changes,
             then throw the exceptions below.
6      if (EventData.Count == 0) // the user didn't specify any event data
7      {
8          //throw new ArgumentException("The data list passed to BuildEventDictionary was empty.");
9      }
10     else if (EventData == null) // the event data was null
11     {
12         //throw new NullReferenceException("The data list passed to BuildEventDictionary was null.")
               ;
13     }
14     else
15     {
16         foreach (AnalyticsData d in EventData)
17         {
18             eventDataDict.Add(d.EventName, GetReflectedData(d.EventInfo));
19         }
20     }
21
22     return eventDataDict;
23 }
```

B.2.4 GetReflectedData Method

```
1  public object GetReflectedData(AnalyticsMemberInfo m)
2  {
3      if (m == null)
4      {
5          throw new NullReferenceException("MAComponent: An AnalyticsMemberInfo passed to
               GetReflectedData was null.");
6      }
7
8      Type t = Type.GetType(m.QualifiedName);
9
10     Component c = gameObject.GetComponent(t); // NOTE: Will only get the first Component found. TODO
           : More work needed to handle duplicate components - this is currently infrequent, and
           therefore unsupported
11
12     if (m.MemberType == MemberTypes.Field) // the member is a field
13     {
14         FieldInfo fi = c.GetType().GetField(m.Name); // construct a FieldInfo from the Component
               Type and field Name
15         if (fi.GetValue(c) == null)
16         {   // warning: the reflected field value was null.
17             // this may be expected or normal behaviour (e.g., the property/field is an object
                   reference)
18             Debug.LogWarning("MAComponent: GetReflectedData FieldInfo was null - returning \"null\""
                   );
19             return "null";
20         }
21         else
22         {
23             if (fi.FieldType == typeof(sbyte))
24             {
25                 return (sbyte)fi.GetValue(c);
26             }
27
28             /* several else-if statements which return the result of an explicit cast have been
                   removed for brevity */
29
30             else if (fi.FieldType == typeof(bool))
31             {
32                 return (bool)fi.GetValue(c);
33             }
34             else // the field is not a numeric, floating-point, decimal, or boolean type
35             {    // so, return its string representation
36                 return fi.GetValue(c).ToString();
37             }
38         }
39     }
40     else if (m.MemberType == MemberTypes.Property) // the member is a property
41     {
```

```
42          PropertyInfo pi = c.GetType().GetProperty(m.Name); // construct a PropertyInfo from the
                Component Type and property Name
43          if (pi.GetValue(c, null) == null)
44          {   // warning: the reflected field value was null.
45              // this may be expected or normal behaviour (e.g., the property/field is an object
                    reference)
46              Debug.LogWarning("MAComponent: GetReflectedData PropertyInfo was null - returning \"null
                    \"");
47              return "null";
48          }
49          else
50          {
51              if (pi.PropertyType == typeof(sbyte))
52              {
53                  return (sbyte)pi.GetValue(c, null);
54              }
55
56              /* several else-if statements which return the result of an explicit cast have been
                    removed for brevity */
57
58              else if (pi.PropertyType == typeof(bool))
59              {
60                  return (bool)pi.GetValue(c, null);
61              }
62              else // the property is not a numeric, floating-point, decimal, or boolean type
63              {    // so, return its string representation
64                  return pi.GetValue(c, null).ToString();
65              }
66          }
67      }
68      else
69      {
70          throw new NotSupportedException("MAComponent: GetReflectedData can only operate with
                MemberTypes.Field or MemberTypes.Property types.");
71      }
72  }
```

B.3 AnalyticsManager.cs

B.3.1 Notify Method

```
1  public static void Notify(string methodName, string className)
2  {
3      AnalyticsComponent[] components = Object.FindObjectsOfType<AnalyticsComponent>(); // find all
           AnalyticsComponent as Component
4
5      foreach (AnalyticsComponent c in components)
6      {
7          // the AnalyticsComponent target matches this method + classname - call the Component's Send
               () method
8          if (c.TriggerMethod.MethodValue == className && c.TriggerMethod.MethodKey == methodName)
9          {
10             Debug.Log("MAManager: Notifying object " + c.name.ToString() + " to fire its Send()
                   event");
11             c.Send();
12         }
13     }
14 }
```

B.3.2 TransmitEvent Method

```
1  public static void TransmitEvent(string eventName, Dictionary<string, object> eventParams,
       GameObject sender)
2  {
3      if (!AnalyticsSettings.Settings.Offline)
4      {
5          LogResult(Analytics.CustomEvent(eventName, eventParams), eventName, sender);
6      }
7      else // always DebugLogResult unless explicitly set to !offline
8      {
9          DebugLogResult(eventName, eventParams, sender);
10     }
11 }
```

B.3.3 LogResult Method

```
1  private static void LogResult(AnalyticsResult r, string eventName, GameObject sender)
2  {
3      Debug.Log("MALogResult: " + sender.name.ToString() + " sending event " + eventName + " returned
           code " + r.ToString());
4  }
```

B.4 AnalyticsFunctions.cs

B.4.1 CleanPropertyString Method

```
1  private static readonly string[] StripStrs = { "\n", " ", "\'", "\"", "\0", "\a", "\b", "\f", "\r",
       "\t", "\v" };
2
3  public static string CleanPropertyString(string propertyString)
4  {
5      foreach (string str in StripStrs)
6      {
7          if (propertyString != null)
8          {
9              if (propertyString.Contains(str))
10             {
11                 propertyString = propertyString.Replace(str, string.Empty);
12             }
13         }
14     }
15
16     return propertyString;
17 }
```

B.5 AnalyticsData.cs

```
1  [Serializable]
2  public class AnalyticsData
3  {
4      public AnalyticsData()
5      { }
6
7      [SerializeField]
8      private string eventName;
9
10     public string EventName
11     {
12         get { return eventName; }
13         set { eventName = value; }
14     }
15
16     [SerializeField]
17     private AnalyticsMemberInfo eventInfo;
18
19     public AnalyticsMemberInfo EventInfo
20     {
21         get { return eventInfo; }
22         set { eventInfo = value; }
23     }
24
25     [SerializeField]
26     private int index = 0;
27
28     public int Index
29     {
30         get { return index; }
31         set { index = value; }
32     }
33 }
```

B.6 AnalyticsEvent.cs

```
1  [AttributeUsage(AttributeTargets.Method)]
2  public class AnalyticsEvent : Attribute
3  {
4      private string description;
5
6      public string Description
7      {
8          get { return description; }
9          set { description = value; }
10     }
11
12     public AnalyticsEvent(string desc)
13     {
14         Description = desc;
15     }
16 }
```

B.7 TestMonoBehaviourScript.cs

```
1  using UnityEngine;
2
3  public class TestMonoBehaviourScript : MonoBehaviour
4  {
5      [AnalyticsEvent("An event which is fired immediately after the game begins")]
6      public void OnTestGameEvent()
7      {
8          AnalyticsManager.Notify("OnTestGameEvent", "TestMonoBehaviourScript");
9      }
10
11     void Update()
12     {
13         if (!run) // only run the test event once
14         {
15             OnTestGameEvent();
16             run = !run;
17         }
18     }
19 }
```

B.8 TimingFunctions.cs

```
1  using System.IO;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.Analytics;
5
6  public static class TimingFunctions
7  {
8      private static readonly string timedNotifyPath = "C:\\timedNotify.csv";
9      private static readonly string timedCEPath = "C:\\timedCE.csv";
10     private static StreamWriter notifySW = new StreamWriter(timedNotifyPath, true); // true ==
           append
11     private static StreamWriter ceSW = new StreamWriter(timedCEPath, true); // true == append
12     private static System.Diagnostics.Stopwatch w = new System.Diagnostics.Stopwatch();
13
14     public static void TimedNotify(string name, string type)
15     {
16         w.Start();
17         AnalyticsManager.Notify(name, type);
18         w.Stop();
19
20         Debug.Log("TimingFunctions: TimedNotify call took " + w.ElapsedMilliseconds + " ms");
21         notifySW.WriteLine("timednotify\t" + w.ElapsedMilliseconds);
22         notifySW.Flush();
23
24         w.Reset();
25     }
26
27     public static void TimedCustomEvent(GameObject o)
28     {
29         AnalyticsComponent c = (AnalyticsComponent)o.GetComponent(typeof(AnalyticsComponent));
30
31         w.Start();
32         Dictionary<string, object> dict = new Dictionary<string, object>();
33         /* code to populate dictionary from GameObject goes here - must be modified for every test
               procedure */
34         Debug.Log("TimingFunctions: TimedCustomEvent returned code " + Analytics.CustomEvent(c.
               EventName, dict));
35         w.Stop();
36
37         Debug.Log("TimingFunctions: TimedCustomEvent call took " + w.ElapsedMilliseconds + " ms");
38         ceSW.WriteLine("timedcustomevent\t" + w.ElapsedMilliseconds);
39         ceSW.Flush();
40
41         w.Reset();
42     }
43 }
```

Appendix C

Project Poster