Event-Based Data Collection Engine for Serious Games

Amith Tudur Raghavendra

Event-Based Data Collection Engine for Serious Games

Amith Tudur Raghavendra

Carnegie Mellon University
Entertainment Technology Center
700 Technology Drive Pittsburgh, PA-15219 310-910-4274
amith@cmu.edu

Abstract. Games with a purpose other than entertainment can be called Serious Games. In this paper, we describe a generic event-based Data Collection Engine (DCE) that has been developed for Serious Games on the Unity Game Engine. Further, we describe a framework that allows for the manipulation and feedback of the collected data back into the game in real-time. The player experiences the visuals, sounds and the game itself that is streamed over the web. The player engages with an enriching, multimedia experience allowing him/her to be immersed in the game. By suitably designing the serious game we could determine the behavior of the player in real world under the given scenario or other scenarios. The DCE is optimized to collect relevant data streamed online without affecting the performance of the game. Also, the DCE is highly flexible and can be setup to collect data for any game developed on the Unity Engine.

Keywords: Unity Game Engine, Serious Games, Data Collection Engine.

1 Introduction

We have developed a system that can collect data from Serious Games[1], while they are being played. The event-based data collection is efficient and can be integrated into any game that has been developed on the Unity Game Engine [2]. Data collection from a Serious Game played online can quickly become immense and voluminous. It would need an efficient approach that would stream both the game as well as the collected data. It would have to be flexible enough to include the different kinds of games that can be created and different data that might have to be collected on a single game. In the sections below we describe a generic event-based data collection approach that can be used to track and analyze data in any Serious Game developed on the Unity Game Engine. The system is designed to track only the data that is relevant, optimizing the data flow for the game played on a browser. Additionally the DCE has the capability to manipulate the collected data and feed it back to the game in real-time. So a game can be customized dynamically to a player based on the previously collected data. The DCE could be used to single out and profile a player over multiple games.

2 Scenarios

Scenarios in Unity are made up of game objects. The game objects could consist of the multimedia like sound spots, animations, models and characters that are interactive. Figure 1 shows one such scenario. The game objects are circled red. Each of these game objects are made up of parameters that determine their state at any instant in the game. The intuition behind the DCE would be the fact that by logging and correlating the states of the parameters at a particular instant we could analyze the scenario in several different ways. Let us take the example of a Market Scenario. We have vendors along the sides of a road. The player has to drive a car through the Market streets. Here the game objects would include the vendor, the items they are selling, road, pavement, car and the driver (player). The player's parameters would include position, view of the market, sound being heard, etc.

Fig. 1. A Scenario in the Unity Game Engine

3 Generic Data Collection Concept

Generic Data Collection is based on the concepts of Condition, Selection and Events. As mentioned earlier it relies on the intuition that by tagging the data suitably, the parameter values can be correlated later and the scenario as a whole could be analyzed to any depth of complexity.

3.1 Condition

Time is abstracted into the concept called a *Condition*. A *Condition* determines *when* a parameter(s) would be logged. The *Condition* can be a relation between multiple objects that may turn out to be true in one or more frames in the game. For example, in the market scenario if we want to know about the case of a reckless driver, the condition would be track *when* position of the car is on the pavement and the velocity of the car is greater than 40mph.

3.2 Selection

The parameters to be logged are abstracted into the concept called a *Selection*. A *Selection* determines *what* would be logged. For example, in the market scenario, suppose we wanted to log the positions of the vendors when the car runs over the pavement and the velocity/path of the car. Then, the parameters to log would be the positions of the vendors, position of the car and the velocity of the car.

3.3 Event

By combining a *Condition* and *Selection* together we get an *Event*. *Events* greatly reduce the complexity of data collection in a world composed of hundreds of objects continually interacting with each other. We check for an *Event* in every frame of the game. An *Event* occurs when a *Condition* is found to be true, in which case we log the *Selection*.

Figure 2 shows the generic concept diagram for the Event handling. *LogEventChecker* looks for an event every frame. This could be customized to any granularity based on the capability of the hardware at hand. However if the data is collected, say every 24 frames, then the trade-off would be the precision of data analysis later on. Once the *LogEventChecker* finds that a given *condition* is satisfied, it triggers a *selection* to be logged. The parameters are collected in a set to avoid redundancy. Finally, the set of all parameters for this frame are logged onto the database.

For example, in the market scenario, an event is composed of the condition and selection that were mentioned above. Hence, when the car runs over the pavement, the condition is met and hence a log is created with the parameters in the selection. The parameters are tagged by a unique gameid and a frame number. Hence the primary key in the database would be (gameid, frame number). The correlation would involve querying the values of the parameters at a particular frame number. In this case we could query for the values of the parameters when the car went over the pavement. This gives us an idea about the game when the event under consideration happened.

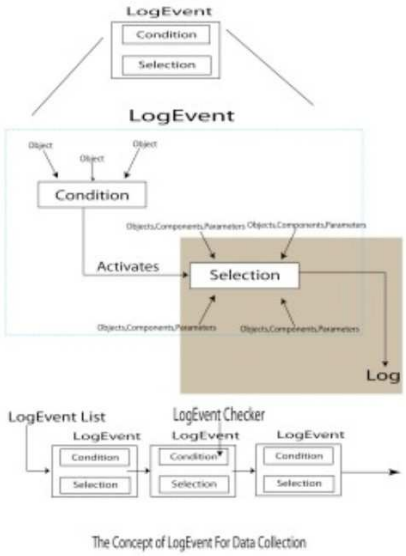

Fig. 2. Concept of Generic Data Collection

3.4 TARDA

Test Application for Reactive Data Analysis (TARDA) is a module attached to the database that feeds the data back into the game changing it in real time. The user could write the relevant analysis logic [3] in the TARDA by utilizing the huge amount of data that is present in the database as a result of the players' previous games. For example, let us take the market scenario again. The Game ID would be a unique ID defining the particular game instance for a particular user and can be used to track a person over multiple sessions of the game. If a particular driver is noted to be reckless over a bunch of games, let's say 20, then we could modify the behavior of the vendors in the 21st game. We could program them to walk away from the car as it approaches. As soon as the car moves over the pavement, the vendors can take the path that the driver is least likely to take. The least likeliness could be determined by using the path data of the car that has been collected from the past 20 games. Thus the game can be programmed to change dynamically by getting its feed from TARDA. The intelligence of the person playing the game can be propagated to the system through TARDA over time. We could mine the data across multiple users over thousands of games that are played by them. For example, we could figure out the expected path of car from the past 2000 games from the most recent 100s of users.

4 The Data Collection Framework

Figure 3 shows the Data Collection framework which includes the implementations for the *EventChecker* and TARDA. We create the game on the Server establishing the

different parameters that would have to be collected at different instants. A customized and streamlined GUI has also been developed to facilitate the Event Creation. Multiple Events that are created in the process are stored on an *EventFile* on the Server. The build is then published and deployed on a web browser. A player could play the game from any part of the world since the game just requires a Unity plug-in. The player can be uniquely tagged over multiple sessions of the game, as data about him/her is gathered by the DCE. All the while, the player is oblivious to the fact that the DCE is working behind the scenes. TARDA that is attached to the Database can be used to tweak the game in *real-time* based on the user's profile that has been gathered over a period of time and this game as well. The Database has multiple tables that are used to store the data. The DATALOG table stores the (key, value) pair that is being logged. TORUN is the database that keeps track of the TARDA commands. These commands can be fed into the game while its running. The game will change its state depending on the command that has been fed in.

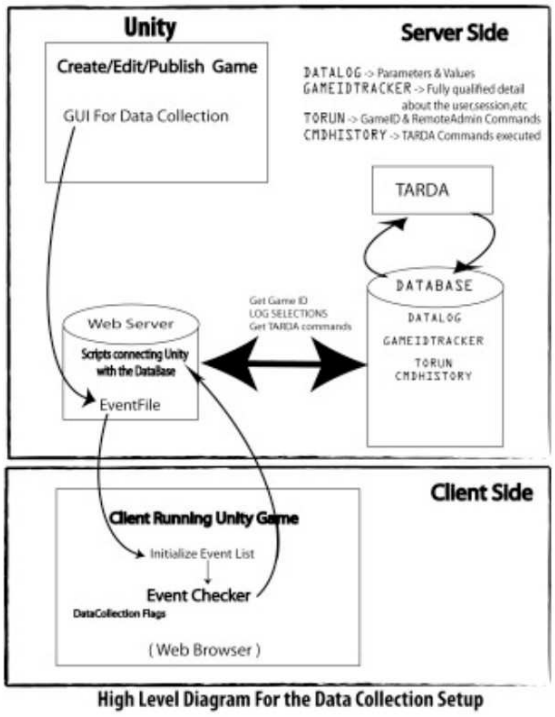

Fig. 3. The Data Collection Framework

5 Implementation Examples

5.1 Editor GUI

Figure 4 shows the game developer building the events with the customized GUI that has been created. The creator goes through the logical process of creating a condition and selection here. Condition is a code in the native language of the game being created for which there will be no learning curve. As it can be seen here, we have selected a person. The person can be referenced using the code that is written in the GUI. Similarly we could have a bunch of objects to be logged when this condition is met. The process is repeated for every event that the developer intends to track. The code written here is evaluated dynamically at runtime. Hence this also provides a mechanism for TARDA to introduce new methods at runtime into the game.

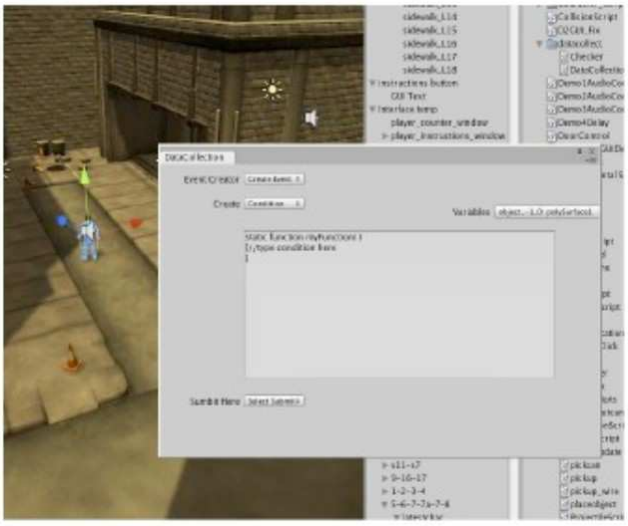

Fig. 4. Create the Events

5.2 Position Tracking

Figure 5 shows a simple example of a game involving a person running in a virtual world collecting ammunitions. This is the top view (map) of the world. The DCE was added to it seamlessly and the person's positions as well as the places in map where he collected the ammunitions were collected into the database.

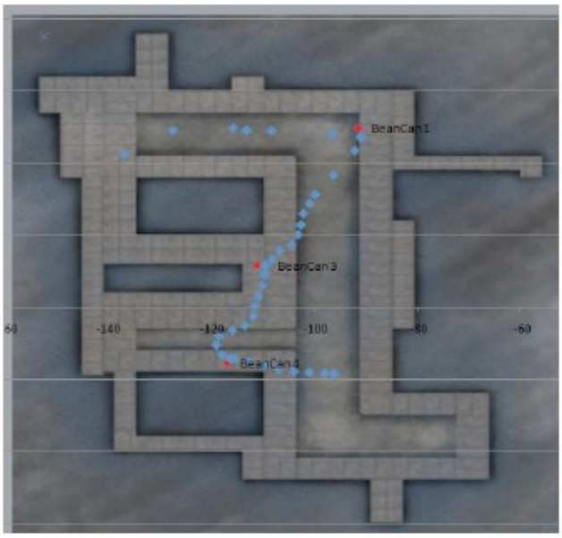

Fig. 5. Position Tracking from the Database

5.3 TARDA Feedback

Figure 6 gives an example of TARDA where we have the person's position being fed back from the database. Game ID of the current game is being shown as well. TARDA has access to the entire database; hence the feeds could be from any of the game that this player or any other player has experienced.

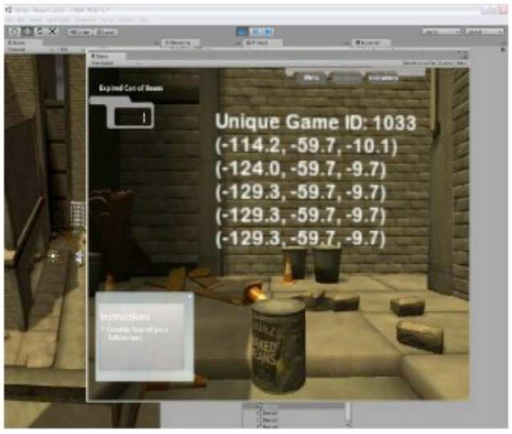

Fig. 6. TARDA feedback into game

Figure 7 gives an example of another TARDA feedback. Here the lines and arrows are pulled into the current game from the database. These lines represent the path taken by this player 10 games before. TARDA feedback is dynamic and lines from

games being played by different users can be manipulated independently and concurrently fed into this game.

Fig. 7. TARDA feedback into game

5.4 Dynamic Tracking

The lines in Figure 8 show the position of a person/car over time. The yellow line is being drawn dynamically as the game proceeds. The black lines represent the path from the previous games. This is a simplified view of the data that can be gathered from the game.

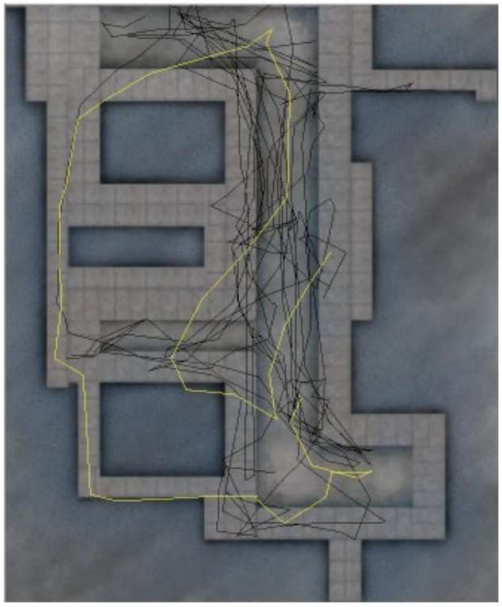

Fig. 8. Position of the player over time

6 ACKNOWLEDGMENTS

My thanks to John Conomikes and Matthew Crozier for being part of the work. Drew Davidson, Mike Christel and Scott Stevens for the guidance. Thanks to Don Marinelli, Gary Sarkesian, Adam Pierce, Shirley Saldamarco and James Burke Jr. for the support. Most importantly, thanks to Austin Jephson, Srinavin Nair, Ethan Rupp, Akash Joshi and Hee Jun Kim for being part of our development team.

References

1. David Michael, Sande Chen. Serious Games, Thomson Publishers.2006.ISBN:1-59200-622-1.
2. Unity Game Engine. http://unity3d.com/
3. Luis von Ahn, Laura Dabbish, 2008. Designing games with a purpose. ISSN:0001-0782, Communications of the ACM, Volume 51, Issue 8. http://doi.acm.org/10.1145/1378704.1378719