Profiling Players Through In-Game Animations

Miguel Marques
Instituto Superior Técnico,
Lisboa, Portugal
miguel.garcia.marques@tecnico.ulisboa.pt

***Abstract*—The purpose of this thesis is to provide a novel method of data collection in video-games, specifically the utilisation of animation data for recording and categorising player behaviour. We begin by explaining the meaning of playtesting and modelling, and we define the concept of animation. We then explain how we extracted the animation data from an already released game that did not provide such functionality. We also approach our implementation of a front-end for the visualisation and analysis of the previously extracted data, as well as some use cases that include both modules. These use cases focus on distinguishing players through the animation data they provide. Finally, we conclude that animation collection is a beneficial method for data collection, and that it can be applied to future games through the development of an add-on for popular game engines.**

***Index Terms*—Animation collection, User modelling, Data collection**

I. Introduction

Electronic games have been around for nearly half a century. In the last twenty years, this industry has grown tremendously and shows no signs of slowing down [4]. Companies and game studios continuously pour their resources into designing games that can satisfy their intended audience, therefore, in order to ease the process of building games, the concept of data collection has been heavily merged into game design practices [8]. Thanks to the widespread access to the internet, it is possible to evaluate design decisions even after the game is released, by having it send data to the developer. This data is most often related to how the player interacted with the game. This interaction is often mentioned as player modelling or player profiling, and it intends to build a virtual representation of the player. It includes a set of techniques meant to model the player through their behaviour, interaction with the game or personality [10]. This resulting model can have several applications, such as adapting gameplay to the user or evaluating the target audience for a game.

In order to accurately profile a player and understand their interactions with the game, a game designer may choose to record the entire play session on video and review it afterwards. While this method guarantees that no information is lost, it is also extremely costly in both computer resources and time spent. Having to manually tag particular scenarios and interactions may take many hours, especially when expanding this process to a myriad of players. As such, many alternatives have been sought out [5] [11]. These alternatives, however, often fail to provide detailed information, working only in small scenarios and not describing the whole picture.

Our interest lies in exploring alternative means of player profiling. We want it to be easy to integrate in the current game design workflow, and light enough on resources so that it does not degrade the player experience when applied to versions of the game that are close to the final retail version. We will explore this in the realm of a player's avatar [7]. An avatar is the player's representation in-game. By looking at their avatar as an expression of the players themselves, and analysing that avatar's behaviour, we can then use that behaviour to profile the player who controls it. We do this by recording the avatar's animations and their duration over time. Throwing a grenade, performing a quick sword attack or a slow and methodical hammer pound, raising a shield, recovering from a hit by an arrow, all of these animations carry meaning with them.

Through this work, we provide a complementary approach to player profiling, by presenting the novel concept of AB2P (Animation-Based Player Profiling). We design a system around the AB2P concept and we demonstrate how to apply it to an existing commercial game.

We will begin by describing key concepts that are relevant for this work. This encompasses the definition and utility of playtesting, a description of current user modelling methods, an explanation of what animations are and how they work in modern games and a summary of what data collection is and how it is performed today. Afterwards, we will explain the complete model of AB2P, with all of its modules. We will then show how we managed to extract animation data from an existing game. This will include our requirements for the module, how we implemented it, and how we tested it. We will then describe how we stored and crunched the extracted data into a user-friendly format through an HTML front-end. Again, this will be complete with its requirements, implementation, testing and evaluation. Having explained how the system works, we will provide a number of use cases where AB2P is at their core. Lastly, we will draw our conclusions from this document and provide pointers for future work within this area.

II. Related Work

A. Playtesting

Playtesting refers to having people play the game in order to guide game design. Through this process, designers are then able to assess if the game is headed towards the direction both them and the players expect [2]. These tests are usually done with a small portion of the intended audience, and are meant to gather feedback and analyse if the design intent for

the game is being fully achieved. The tests are also done in phases, typically named as pre-alpha, alpha and beta tests [6]. This gives the developers several opportunities to fine-tune the game to their desire, as well as address potential flaws before release.

In this step of game development, data collection is crucial [2]. By collecting data and viewing it in a readable format, developers are able to pinpoint issues with the game without having to go through full video reviews of the playtesting. Collecting animations comes, then, as an addition to the current data collection methods during playtesting, aimed at providing clear and understandable information about the player and how they interact with the game.

B. Animations

In modern games, an animation is portrayed by a change of state in an object. This change involves movement or some form of distortion applied to that object. Animations are usually done in specialised software for them, such Autodesk Maya or Blender [3]. In video-games, animations are split into programmable pieces. As an example, consider a player character waving their hand. This particular sequence of raising the arm, opening the hand, rotating the elbow and returning to the default position is considered to be an entire animation, with its own name and duration [3].

When imported into a game engine, animations may then be applied to the player character whenever the designer wants it to. Both the animation name and duration are relevant to this work. The animation's distinct identification allows us to know exactly how the player behaved on any specific scenario. Its duration is also useful, as we are then able to tell how much time the player spent doing that animation over a larger period of time.

Another important concept to take notice of regarding animations is that of animation cancelling. An animation does not have to be performed in its entirety until it is stopped. This happens very often in faster-paced games, where the player may be in the process of swinging a sword and suddenly performs a dodge roll to avoid incoming damage.

C. Data Collection

Data collection is a process where input from the interaction between a user and a program is recorded for further analysis. The data that is collected is often meant to evaluate how close the user's experience is to what is intended [1]. Animation collection can fit into the current methods of data collection as another source of useful information. By turning data from animations into something identifiable and measurable, we can then produce accurate assumptions on how the player behaves.

III. AB2P Model

AB2P can be defined as a system for analysing player behaviour by recording and categorising the animations of their in-game avatar over a period of time. The overall architecture of AB2P can be seen on Figure 1. From a high level point of view, AB2P can be split into two separate modules. The first module, aptly named Animation Collector, handles the collection and storage of animation data from a running game. This means that this module is active while the player is engaging with the game itself.

This animation data is then stored at an arbitrary interval of time (timeframe). Every N seconds, all the animations that the player performed and their duration within those N seconds will be stored in a file. It is important to define this timeframe first, as it will determine how precise the recorded data will be. The end result is a time-ordered set of timeframes of animation data, with a level of detail relative to the chosen size for the timeframes.

The second module in our system is the Animation Visualisation module. The purpose of this module is to read the collected animation data from the file system, upload it into a database and display it on a front-end, in a visually understandable format. To facilitate the understanding and handling of animation data from the user's point of view, the concept of categorisations is also part of this module. Categorisations allow the user to group animations into categories and view a larger picture of how the animations happen over time.

There is one additional component called Animation Tagger. If a game is built with AB2P in mind, it is very simple to read the animation names from the source files. However, in order to apply AB2P to an already released commercial game, as is the case with our work, it means that there is no direct access to the animation names, and all the animations are instead defined by a unique number with little meaning to the end user. To counteract this, our Animation Tagger reads that animation number, plays it in the game and asks us to give it a name.

IV. Animation Extraction

A. Requirements

We will now lay down some of the most important requirements that guided our implementation and subsequent evaluation.

1) Animation Polling: Our most basic requirement is the ability to query a game for the current animation that the player's avatar is going through. Furthermore, we must ensure that this query does not crash the game, the animation extraction app or cause unexpected behaviour in-game. This generally means that we want to gracefully handle scenarios where the current animation is null, such as when the game is starting or loading different scenes.

We also need to poll for the current animation on as many frames of gameplay as possible, in order to obtain the full spectrum of animations throughout a session. Game frame-rates are generally locked at either 30 or 60 frames per second. Therefore, we require the ability to poll for the current animation for at least 30 times per second with no noticeable slow-downs on the game itself. Slow-downs would both degrade the player experience and the quality of the data we gather.

2) Animation Storage: Our app must be able to store the collected animations into the file system every N seconds (timeframes), configurable by the developer. Writing to the file

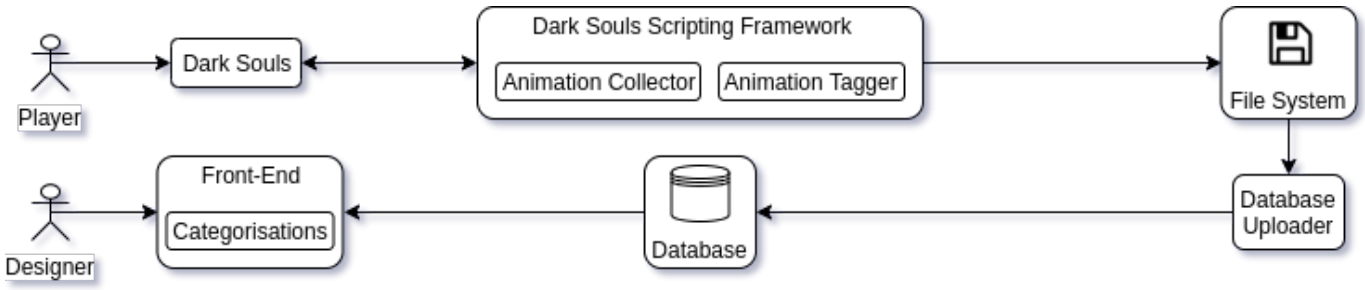

Fig. 1. Architecture of the AB2P model.

system on every frame would be overly taxing, and writing only at the end of a session could mean losing the entire session if any crash were to occur. Furthermore, we want to be able to split the animation data into time-ordered chunks, so that we can visualise how it varies over time. Any files that are being written to by the app must be previously locked, as we expect that a data visualisation app could be reading these files at any time. The data must also be exported in JSON, which is a common format that most programming languages can easily parse. Finally, we want this data to be as lightweight as possible, both to avoid overloading the system's storage and to facilitate its transfer.

3) Session Management: The app must be aware of the user's session. Specifically, this means that the app must understand which avatar is being played and store the collected animation data into that avatar's session. Since different people will likely play different avatars, we must ensure that the animation data from one avatar will not be mixed with the data from another. Furthermore, the app must support resuming a previous session, as the player may wish to stop playing and close their game, so that they may continue later on. This entails detecting the game's closure, knowing the timeframe at which the last session ended, and resuming data collection from that timeframe onward when it is opened again.

4) Independent Execution: Tools that read data from another app's memory generally require that the targeted app is open beforehand. However, as games tend to occupy the entire screen, having to open the game and then move away from it to open the tool will likely prove to be cumbersome for the end-user. Therefore, we require that our app is capable of being opened before the game, staying quiet and idle until the targeted game is detected.

Another requirement is that the app does not need any sort of input from the player, other than opening it. Detecting the game, naming the session and performing the animation collection must be an entirely automated process. Finally, we require that the app can handle collecting data for long periods of time with no crashes, as players often stretch their sessions for multiple hours.

B. Implementation

1) Dark Souls: Before moving into the details of how we implemented our model, we will begin by describing the game we have chosen, Dark Souls. Dark Souls is a digital game that was first released for the PlayStation 3 in 2011. A PC version came out later in 2012. The game is played on a third-person view. The player controls a character that was left to die in a cell but somehow finds a way out. Dark Souls employs the usage of animations very seriously. The wrong movement at the wrong time will most often cause death or serious injury to the character. This means that any animation the player performs carries plenty of meaning, just like the animations the player is forced to partake in, such as recovering from an enemy's blow.

2) Dark Souls Scripting Framework: Some games provide direct access to their data through an API or similar tool. Dark Souls is not one of those games. While this makes the process of extracting relevant data more difficult, it does not make it impossible. Thankfully, we can rely on community-built tools that help us in the process of gathering data. In October 2017, a community member nicknamed Wulf2k released the Dark Souls Scripting Framework [9] (DSSF). This framework allows us to interact with the game's variables and code scripts around them using Python.

3) Extracting the Animations: The function for reading an animation was already implemented by the framework. This function, however, would crash the app if it was called during the loading of a character. We worked around this issue by checking the player's name before checking their animation, as that function would simply return a null value when a character was being loaded. In regard to polling, we requested the animation ID for the player every 1/30th of a second, which is one frame of this game. We then relied on a Python dictionary to store the animation data. The end result is a dictionary with the animation IDs as keys and the number of frames where they were detected as values.

Every 5 seconds, the data from this dictionary is stored into a distinct JSON file named after the current timeframe. This allows us to analyse how these animations varied over time and prevents corrupting data by regularly recording to the disk what is in memory.

Following the successful recording of animations, we then proceeded to implement session handling. As we do not have direct access to the save data, we used accessible variables to generate session names. When a character loading is detected, a folder is created with that specific naming scheme. If this

folder already exists, it is used instead. This allows sessions to be continued over time.

Regarding error handling, there is a fair amount of checks that must be done before execution. Namely, detecting whether Dark Souls is open or not and whether a character has already been loaded or not.

C. Evaluation

In order to test the app, we sought out multiple users and had them play the game for extended periods of time. Some of these played more than one character, which meant multiple sessions. The total number of unique users was 7. The total number of sessions was 12. We gathered over twelve hours of gameplay from these users and over 160000 frames of animation. All the testing below was done on computers that could handle an unmodified version of Dark Souls at 30 frames per second. The system specifications varied over the users, with the lowest speed CPU being an Intel Core i5-3320M (4 cores, 2.6GHz each) with an integrated GPU.

1) Animation Polling: Querying for the current animation was made possible thanks to DSSF. For the app's reliability, our final test users did not experience any crashing or unexpected behaviour. In terms of frame-rate and polling speed, there were no issues with 30 queries every second. On system resources, the app hovered 22 megabytes of RAM usage and less than 1% CPU usage.

2) Animation Storage: When it came to storing the animations over time, the process went smoothly. The files were correctly exported as JSON, with little over 2% disk usage on average. The overall size of the 160000 animations that we gathered weighed about 330KB, which was a very welcome success.

3) Session Management: We had several people playing their characters over time, each with their own character on their computer, and some playing on the same computer but on a different avatar when their turn was up. The app had no issue distinguishing each session, as well as resuming it when necessary. One flaw is that we cannot access the game's save data directly, therefore we rely on a combination of the avatar's name, their class and the computer's name to assess if the current avatar is new or should be resumed. If the player decides to delete their character and create a new one with the exact same name, same class and playing on the same computer, our app will assume that they are resuming a previous session, resulting in inconsistent data. This would not have been an issue if we had proper access to the game's internal data.

4) Independent Execution: The app can be run before the game is open, and it will stay idle, polling for the game's executable in the process list until it can find it. Once found, it hooks onto the game and begins collecting the animation data. This entire process does not need any input from the player, as per our requirements. We also had users play the game for over an hour at different occasions, and there were no crashes.

V. Animation Visualisation

A. Requirements

Our data visualisation module can be split into three smaller modules: uploading the animation data to a server, tagging the animation IDs with readable names and, finally, viewing the data itself. We will now establish the requirements for each of these smaller modules.

1) Uploading the Data: We need the data uploading app to be capable of uploading the entire session reliably. This means that a faulty internet connection, an app crash or the user quitting the app should all be taken into account. We also want to free the user from any form of input. Finally, the data uploading app must be capable of running and uploading data in parallel with the animation extraction app.

2) Tagging the Animations: In an ideal scenario, this module is not needed. In our circumstances, it becomes a requirement. The module must be capable of detecting that a new animation was played in-game and subsequently ask the user to give it a name. It must also allow us to play-back an animation in-game by reading the respective animation number from a file.

3) Viewing the Data: We require the ability to group animations into different categories. On the front-end, we must be able to select an available session from the database, select a categorisation and view how each category of animations varied over time for that session.

B. Implementation

1) Database Uploading App: Our app for uploading the animation data to a database is built in Python 3.6.

Our first step is checking if there is any session available for uploading. If a new session is found, a new unique ID is generated for that session. If the detected session already exists, the ID from the database is used instead. Uploading is done by finding the oldest file within the session folder, reading its contents, which are in JSON, and uploading them to the database. Before accessing the file, a check is done for file locks. Several checks are also done for internet connection. This implementation allows the app to be closed during execution and opened again, with no harm to the files being uploaded.

2) Animation Tagger: We split the tagging of animations into two different tools, both written in Python 3.6. The first tool attaches itself to the DARKSOULS.exe process and detects any animation performed, similarly to the animation extraction app. Whenever an animation that it has not seen before is detected, a terminal window asks for a name. The second tool selects animation IDs from the database that have yet to be mapped to a name and orders them by the time spent doing them. Then, it attaches itself to Dark Souls and plays each animation on the player character, asking the user to name them. Many animations are programmed into the game as a set of smaller animations. As an example, let us consider drinking a potion, named "estus" in Dark Souls. Grabbing the estus from the bag, lifting it and drinking it are three

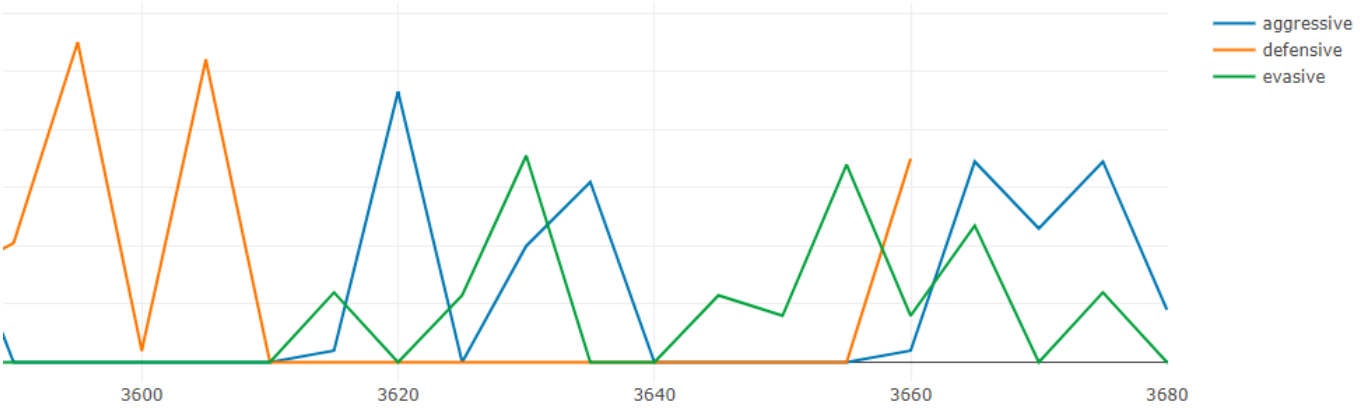

Fig. 2. Categorisation applied to animations, zoomed in on the categories.

separate animations from the game's point of view. However, from the user's point of view, they should be considered a single motion. We marked such animations by giving them very similar names, such as estus_1 and estus_2. A simple function was then run on the data, detecting similar names and giving them a group name. In our previously mentioned example, this now means that all the animations related to drinking estus belong to the same group name "estus", and this group is a sum of all the smaller animation frames combined.

3) Front-End: In order to view all the data that we have gathered thus far, we built a website. It was developed using PHP, HTML5 and Javascript. It provides two ways to look at the data. One of them is by checking a specific session and the other one is comparing up to three different sessions. Upon choosing a session, the user can immediately see some raw data regarding that session. This raw data is a sum of all the time spent on each animation for that session, ordered by said time. The user may then choose a categorisation. Categorisations are defined within a JSON file on the server. Within that file, there are several arrays, each with a category name on it, such as "Aggressive" and "Evasive". Within those arrays, we put the animation names that we believe belong to that category. By setting the categorisations like this, the data only needs to be recorded once, even if no categorisations for it have yet been made. After picking a categorisation, the user will then be presented with a line chart made in plotly.js that displays how many frames the player spent on each category over time. An example of this chart can be seen on Figure 2. Regarding the comparison tool, the user can select up to three different sessions. After selecting them, a bar chart will be displayed. Our goal with this chart is to display where the sessions are similar to and different from each other. To do this, we begin by gathering all the distinct animations that were done by the first session selected, and we order these from the most frames spent on to the least frames spent on. For the two remaining sessions, we select the same animations that were done in the first session and place their duration next to the bars of the first session. We also normalise everything by dividing each session over their total playtime.

C. Evaluation

All of the testing and evaluation described below was done on an Intel Core i5-3320M (4 cores, 2.6GHz each) CPU, with its integrated GPU and 8gb of DDR3 RAM.

1) Uploading the Data: The uploading app is fully capable of being closed down and started up again, resuming its uploading process from where it was cut off. This was tested by first uploading a session entirely, and then uploading that session again, under a different name, and forcibly closing the app at several points in time. Finally, we compared the uploaded data from both sessions and found it to be equal.

2) Tagging the Animations: The tools are capable of replaying animations inside the game, animations can be tagged, grouped and uploaded, all with the use of various different scripts. However, they are not very user-friendly, as they are not supposed to be used often.

3) Viewing the Data: We have a front-end that is live and running. Categorisations have been implemented in their entirety, and the user is free to choose from a set of samples. The chart engine can handle thousands of rows of data with no visible hiccups, and it provides zooming and panning capabilities to the user.

VI. Use Cases

Our goal with these use cases is to showcase how this tool can be used to help inform the process of game design. In particular, we are going to look at how we can compare playthroughs of the same game with different classes, different players and even different content.

A. Scenarios

In regard to evaluating the actual data, we considered a few different scenarios. One of them was comparing gameplay sessions from the same player with a modified (modded) and unmodified (unmodded) version of Dark Souls. The unmodded version of Dark Souls is the regular game as it was designed by the original developers. The modded version, however, is

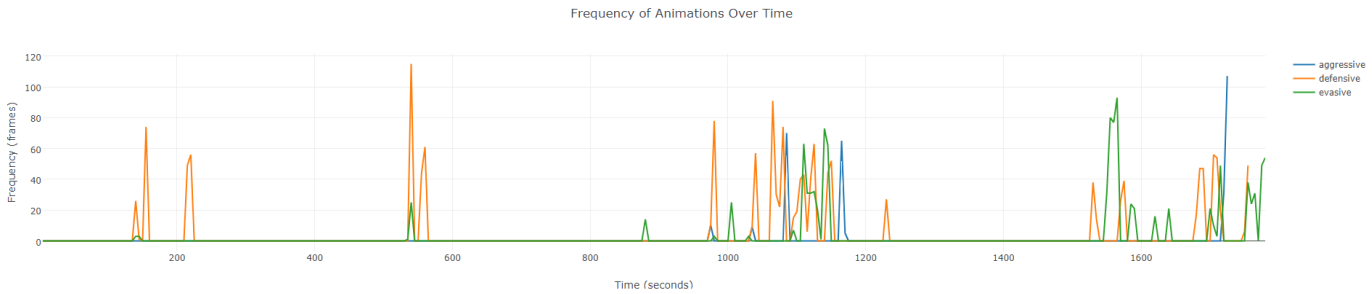

Fig. 3. Aggressive-Defensive-Evasive categorisation for the experienced player.

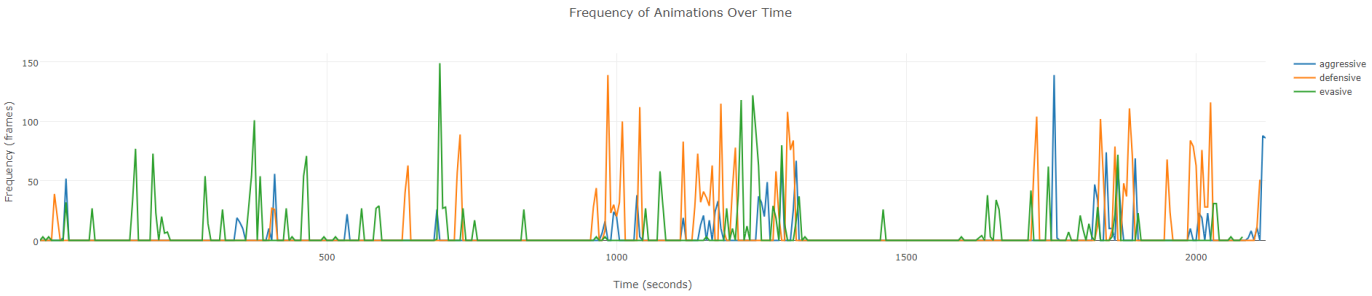

Fig. 4. Aggressive-Defensive-Evasive categorisation for the newbie player.

a re-imagining of the Dark Souls world called "Daughters of Ash". DLC, which stands from Downloadable Content, is a package of new content for a game that developers release later in time. Our goal with this particular experiment was to demonstrate how a mod or DLC can impact the experience in a meaningful way.

Another experiment that we considered was comparing the sessions of two different players, particularly players we could separate as being *pros* or *newbies*. We defined a pro player as someone who understands the game and its mechanics well, and a newbie player as someone who has never played Dark Souls or similar games before.

We also compared sessions from the same players with different starting classes. While the class system in Dark Souls is very fluid, as it does not lock the player into any specific progression throughout the game, the players are given a different set of starting gear depending on the class they chose. We sought to assess if different classes truly influenced the start of the game, as the developers certainly intended.

B. Categorisations

At the moment, we provide two categorisations: Aggressive-Defensive-Evasive (ADE) and Pro-Newbie (PN). These categorisations are not the main aspect of our work, they are a sample of what can be done with this tool.

The ADE categorisation groups all attacking animations into the Aggressive category, defensive animations, such as blocking an attack, into the Defensive category, and evasive animations, such as rolling away, into the Evasive categorisation.

The PN categorisation intends to distinguish skilled players from those who barely ever played the game. Note that this categorisation does not try to label each player as a pro or newbie. Instead, it shows where both players performed pro animations and newbie animations over time. It was developed by analysing the sessions of two players, where one had no prior knowledge of the game, and the other had been playing for several hours already. We then checked which animations were used most often by one player and not used by the other.

The sections below will showcase some interesting comparisons that we came across during our analysis of various sessions.

C. Base Game and DLC Comparison

In this scenario, a player goes through both the base game and the DLC, using the same class.

1) Raw Data: The relevant animations that the user spent most of their time on in the DLC session seem to be generally similar to the ones the user spent on the base game. This is to be expected, as the DLC does not add new mechanics, only new content.

2) Aggressive-Defensive-Evasive: This categorisation allowed us to notice major differences in playstyle for this scenario. In the base game, the player fights far more defensively than on the DLC, and is also hurt far less. This showcases that the DLC may indeed force different playstyles on the player.

D. Different Players

This scenario uses two different players, one that has never played the game before and another that has over 20 hours of gameplay. They use the same class.

1) Raw Data: We saw that these two players use similar animations, though in very different proportions. One major difference we notice is how long the newbie player spent on hurt animations. It is to be expected, as their knowledge of the game is low.

2) Pro-Newbie: We saw that the inexperienced player has several more spikes of newbie behaviour than the experienced player, though the experienced player also has a fair amount of them. This suggests that the current categorisation for pro players might be flawed, as an experienced player should not be displaying signs of being a newbie. Do note that this is just a sample of a categorisation, and creating a more accurate categorisation should require more than the few sessions we have to extrapolate from.

3) Aggressive-Defensive-Evasive: This categorisation provided a very clear distinction in both players. Figure 3 showcases the resulting chart for the experienced player, and Figure 4 displays the same categorisation applied to the newbie player. The newbie player spent far more time on aggressive animations than the experienced player did. They also occupied their time with a fairly chaotic mix of aggression, defence and evasion, whereas the experienced player had clear and concise spikes of action for each category, appearing to be in control.

E. Different Starting Classes

In this scenario, one player is given two different classes to go through a session. The two classes that are being considered are the Pyromancer and the Wanderer.

1) Raw Data: When looking at the raw data for these two sessions, the first noticeable difference lies in the weapon attacks. The Pyromancer class relied on their slow weapon to attack, whereas the Wanderer utilised a fast swinging weapon.

2) Aggressive-Defensive-Evasive: It was noticeable that the average time spent in combat animations for the Wanderer class was higher than for the Pyromancer class. Combat phases also seem to be more spread out on the Pyromancer class than on the Wanderer, implying that the Wanderer either continuously finds new enemies to face or spends more time on them than the Pyromancer. Also noticeable is the overall lack of defence from the Wanderer, when compared to the Pyromancer.

VII. Conclusion

Through this document, we conclude that animation collection is a beneficial method of data collection that is low on resources and allows us to read player behaviour without resorting to reviewing entire recorded sessions on video. It may prove very useful in playtesting scenarios for adjusting gameplay before release, as it models the users in regard to the animations they spend the most of their time on, which can be seen as what a game is about - the animations that happen on screen. All the requirements that we set were met, namely the ability to record data reliably, keep it with a low relative size and display it to the end-user, filtered by categorisations. We also realised that the meaning of an animation can be difficult to pinpoint at times. While some animations are easy to address, such as attacking with a weapon, other animations such as dodging or jumping may be more difficult to understand. Dodging can be used as an offensive or defensive tool, or even as a form of moving faster in some situations. This is relevant because the user of this tool must provide their own meaning to the animations they intend to analyse.

A. Future Work

In the future, it would be very beneficial to integrate this entire concept as an add-on to existing and popular game engines, such as Unity or Unreal Engine. Providing this method of data collection in a hassle-free environment would prove useful to game developers in general. It would also be helpful to provide a user interface for managing categorisations, animations and recorded sessions. Another aspect that could be improved in the future would be to use machine learning for automated categorisations.

References

[1] W. Albert and T. Tullis. *Measuring the user experience: collecting, analyzing, and presenting usability metrics*. Newnes, 2013.

[2] J. O. Choi, J. Forlizzi, M. Christel, R. Moeller, M. Bates, and J. Hammer. Playtesting with a purpose. In *Proceedings of the 2016 annual symposium on computer-human interaction in play*, pages 254–265. ACM, 2016.

[3] W. Goldstone. *Unity game development essentials*. Packt Publishing Ltd, 2009.

[4] A. Marchand and T. Hennig-Thurau. Value creation in the video game industry: Industry economics, consumer benefits, and research opportunities. *Journal of Interactive Marketing*, 27(3):141–157, 2013.

[5] L. E. Nacke, C. Bateman, and R. L. Mandryk. Brainhex: A neurobiological gamer typology survey. *Entertainment computing*, 5(1):55–62, 2014.

[6] R. Ramadan and Y. Widyani. Game development life cycle guidelines. In *2013 International Conference on Advanced Computer Science and Information Systems (ICACSIS)*, pages 95–100. IEEE, 2013.

[7] B. Rehak. Playing at being: Psychoanalysis and the avatar. In *The video game theory reader*, pages 125–150. Routledge, 2013.

[8] R. T. Wood, M. D. Griffiths, and V. Eatough. Online data collection from video game players: Methodological issues. *CyberPsychology & Behavior*, 7(5):511–518, 2004.

[9] Wulf2k. Dark souls scripting framework. https://github.com/Wulf2k/DarkSoulsScripting. Accessed: 2019-05-03.

[10] G. N. Yannakakis, P. Spronck, D. Loiacono, and E. André. Player modeling. In *Dagstuhl Follow-Ups*, volume 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

[11] N. Yee. The gamer motivation profile: What we learned from 250,000 gamers. In *Proceedings of the 2016 Annual Symposium on Computer-Human Interaction in Play*, pages 2–2. ACM, 2016.