Capturing Game Telemetry with Provenance

Troy Costa Kohwalter[1*] Leonardo Gresta Paulino Murta[1] Esteban Gonzalez Walter Clua[1]

Universidade Federal Fluminense, Computação, Brasil [1]

ABSTRACT

The outcome of a game session is derived from a series of events, decisions, and interactions that are made during the game. Many processes and techniques have been developed by the game industry in order to understand this outcome. A successful method is game analytics, which aims at understanding the player behavior patterns to improve game quality and enhance the player experience. However, the current methods for analytics are not sufficient to capture the underlying cause-and-effect influences that shape the outcome of a game session. These relationships allow developers and designers to better identify possible mistakes in the gameplay design or to fine-tune their games. In a recent work, Kohwalter *et al.* introduced a conceptual framework based on provenance to capture these relationships and manually instantiated such framework in some games. In this paper, we propose a concrete component for capturing provenance data and the cause-and-effect relationships among game objects, and for automatically building the correspondent provenance graph. This provenance data allows a more powerful support for the visual game analytics. We implemented our component in the Unity game engine and show two case studies over open-source games.

Key-words: Game, Game Analytics, Tracked Game Data, Provenance Graph.

1 INTRODUCTION

The analysis of tracked game data have become an important stage of game design and production in the last few years [1]. It brings advantages, such as measuring the game stability [2], dynamic adjusting the difficulty of the game [3], performing behavioral analysis [4], balancing the game experience [5], understanding common behaviors [6], and even improving the monetization process [1]. Moreover, game telemetry allows game developers to collect player interactions in the game inconspicuously over extended time periods, during production and after deployment.

However, tracking game data and making it understandable is challenging due to the complexity of the games, leading to huge amounts of information. Moreover, deciding which information should be tracked and recorded is another challenge. One of the most common types of telemetry data is though states changes [7], [8], [9]. Even though state data is easier to examine, they lack contextual information and provides only high-level view of what transpired in the game. In contrast, telemetry data that captures events [10], [11] can provide more low-level and fine-grained information, capturing and describing player activity and relating more closely to the game session. Furthermore, since the data is collected at fine-grain, developers can use aggregating techniques to summarize the data by giving an overview of the game sessions and only digging through the fine-grained data when necessary.

However, no known approaches for game analytics take into consideration the cause-and-effect relationships between events during a game session, which may be an important factor for determining the reasons that led to a certain outcome. In a recent work, Kohwalter *et al.* [12] introduced the usage of digital provenance[1] in games in order to detect these cause-and-effect relationships. The main goal of that work was to propose a conceptual framework, named *Provenance in Games* (PinG), which collects information during a game session and maps it to provenance terms, providing the means for a post-game analysis. This conceptual framework was applied over a game named SDM [14], which focuses on teaching Software Engineering concepts. The provenance support in SDM allowed for a broader range of analysis by using collected provenance information to generate a provenance graph [15]. In another work, Lidson *et al.* [16] extracted provenance information using a non-intrusive technique through image processing mechanisms. In a more recent work, Kohwalter *et al.* [17] also demonstrated the benefits of using the PinG approach during game analysis of serious games, helping students to understand the underlying reasons for an outcome.

The main goal of this paper is to propose a component for capturing the provenance data and automatically generate the provenance graph for analysis. The generated provenance graph can be used for data mining, automatic analysis tools, or a visualization tool, such as *Prov Viewer* [18], a provenance graph visualization tool that supports multiple features for visual data analysis, including spatial-referencing the graph in the game level map. We implemented our provenance capture component in the Unity game engine, making simple the adoption of the PinG conceptual framework by existing games. We present our PinG component in action by applying it over two different games, showing that we are able to capture cause-and-effect relationships and visualize these relationships over the game map for proper visual analysis.

The remaining of the paper is organized as follows: The second section presents related work and the third section provides background information in the form of an overview of the PinG conceptual framework. The fourth section presents our proposed PinG component. The fifth section shows two case studies over different games and the last section concludes this work, pointing out future works.

2 RELATED WORK

The literature adopts different terms for **tracked game data**, such as gameplay data, logged data, play traces, and telemetry data. Moreover, the process of analyzing such data, referenced here as **game analytics**, is also named in different ways, such as gameplay visualization, visual data mining, and game session analysis. In this section, we kept the original terms of each work, as they are usually reflected in the approaches' names.

Joslin [10] proposed the *Gameplay Visualization Manifesto* (GVM), which is a framework for gameplay data logging that uncovers gameplay events by attaching logging methods in game objects responsible for generating relevant events during the game. The event model is the basis for the game data logging framework. It encapsulates the information that is desired by users and

[1]Provenance refers to the documented history of an object's life cycle and is generally used in the context of art, digital data, and science [13].

*e-mail: tkohwalter@ic.uff.br

classifies the events in three groups: immersion, quest, and social. The immersion group represents events related to increasing the player's sensation of being involved in the game flux. The quest group represents events related to quest creation, execution, and analysis. Lastly, the social group represents events related to social factors in the game, such as group meeting or interaction with other characters.

The main application of GVM is for collecting game metrics, such as player deaths, position, time spent in available features (e.g., crafting and fighting), item usage (e.g., equipment), actions performed, and player enjoyment. Therefore, GVM does not track cause-and-effect relationships, only the executed actions along with their timestamp and location, in addition to character attributes and equipment.

Kim et al. [11] proposed the *Tracking Real-Time User Experience* (TRUE) approach that combines human-computer interaction (HCI) instrumentation, which collects *user initiated events* (UIEs), and log file analysis techniques in order to automatically record user interactions with games. Thus, TRUE can capture behavioral data and the attitudinal information behind the decisions made by the player in order to obtain better understanding of the context of each captured behavior.

Nevertheless, the designer still needs to infer the reasons behind the elements that led to an outcome. This occurs because the contextual information are only extra attributes that were tracked during the execution of the action and not actual relationships between events and thus it does not capture cause-and-effect relationships and must be inferred by the designer when analyzing the logged data. Moreover, TRUE was designed for the industry and is not easily available for indie companies. Even though we did not explore attitudinal data with PinG, it can be trivially incorporated in our approach as attributes for the player's actions or by creating specific activity vertices only for the attitudinal data when they are captured.

Playtracer [8], which is a visual tool designed to illustrate how groups of players move through the game space, aids the designer by tracking game states and showing common pathways and alternatives that players used to succeed or fail in their tasks, identifying pitfalls and anomalies in the scene. Nonetheless, *Playtracer* does not consider temporal information and does not preserve the order of the states visited by players when he/she revisits the same state. Moreover, incorporating *Playtracer* in the game design is challenging because it requires designers to define a state distance metric and identify relevant states.

Play-Graph [7] captures and illustrates the sequence of states and the actions that caused the state change from the players over the course of the game. In the Play-Graph context, a game state describes a certain configuration of the game or an entity, while actions consist on player interactions within the game, such as shooting, jumping, or using an object. In this concept, a game is viewed as a finite state machine with a finite number of states and transitions between them. The states are composed of a set of attributes from the game and players trigger actions at some specific points in the game. However, due to the nature of how the data is structured in Play-Graph, the understanding of player behavior is guided by the player progression in the game (*e.g.*, killed a boss), and not by how he/she interacted with the world (*e.g.*, combat rounds from the battle against the boss). From the available documentation, there is no way to determine interactions or influences. Only the changes from one state to another, caused by an action executed by the player, can be identified. Conversely, influences in the player's action, such as an influence from another character that affected the transition of one state to another, are not present in the graph (there are no edges linking edges).

3 PinG: Provenance In Games

The *Provenance in Games* (PinG) conceptual framework [12] was developed to map provenance concepts to the context of games. PinG was based on the PROV model [19], which provides the basis for specifying information that was involved in creating or influencing a particular object. Thus, PinG provides a mapping of elements from the provenance domain to the corresponding elements in a game domain, relating each data type of the provenance graph to typical elements found in games. In the game context, the provenance graph shows actions performed by characters (player or non-player) and events that occurred during game sessions, and the causal dependencies among these actions or events. It is important to notice that the edges' orientation in the provenance graph goes from the present to the past, instead of the common orientation used in graphs, which are from the past to the future. In order to track provenance data, it was first necessary to define the provenance's counterparts in the game context to create a provenance graph to capture and represent a game session through the events that occurred.

In order to use the provenance vertex types, it is first necessary to define their counterparts in the game context. In the context of provenance, *entities* are defined as physical or digital objects. In the PinG approach, they are mapped into game objects without autonomous behavior. In provenance, an *agent* corresponds to a person, an organization, or anything with responsibilities. In the game context, agents are mapped into characters present in the game or game objects with autonomous behavior, such as event controllers, plot triggers, or the game's artificial intelligence overseer that manages the plot. Therefore, *agents* represent elements capable of making decisions or that have responsibilities in the game, while *entities* represent objects with no autonomous behavior. Lastly, *activities* are defined as actions taken by *agents* or interactions with *entities*. In the game context, *activities* are defined as actions executed or events that occurred throughout the game, such as attacking, dodging, and jumping.

The information collected during the game is used for the generation of the provenance graph, which in turn is used by the visualization tool. In other words, the information collected throughout the game session is the information displayed by the provenance graph for analysis. Thus, all relevant data should be registered, preferentially at fine grain. The way of measuring relevance varies from game to game, but ideally, it is any information deemed relevant by the game designer that can be used to aid the analysis process.

4 PinG for Unity

In a previous work, Kohwalter *et al.* [17] implemented the provenance data gathering directly in the game. Thus, in this work we introduce a generic component capable of gathering provenance during a game session, leading to a domain-independent and low-coupling solution. This PinG component for Unity is composed of components written in *UnityScript* (a version of JavaScript used by Unity3D) that provides easier provenance extraction, requiring minimal coding in the game's existing components. This component has three different types of modules: seven *Core* modules, one *Interface* module, and five *Auxiliary* modules.

Figure 1 illustrates a simplified class diagram for this component, named PinGU (PinG for Unity). *Core* classes are in yellow, *Interface* classes are in light blue, and *Auxiliary* classes are in orange. The *Core* classes represent the infrastructure of PinG and are responsible for provenance information management, making everything transparent to the game designer. Analogously, it can be referenced as the provenance "server". Behind the scenes, the *Provenance Controller* class manages the creation of new vertices and edges and links them in the provenance graph. Meanwhile, the

Influence Controller class manages the cause-and-effect relationships (influence edges), dealing with possible influences and passing it to the *Provenance Controller* class when they actually materialize in the game. The *Provenance Container* class exports the data to a XML file.

The *Interface* classes are the gateway between the game and the *Core* classes. While the *Core* classes can be seen as the server, the *Interface* classes can be seen as the client application. The *Extract Provenance* class is where all provenance-gathering operations must pass through in order to reach the provenance-managing unit (or server). The *Auxiliary* classes contain pre-defined functions customized for a specific behavior, making easier to implement the provenance gathering.

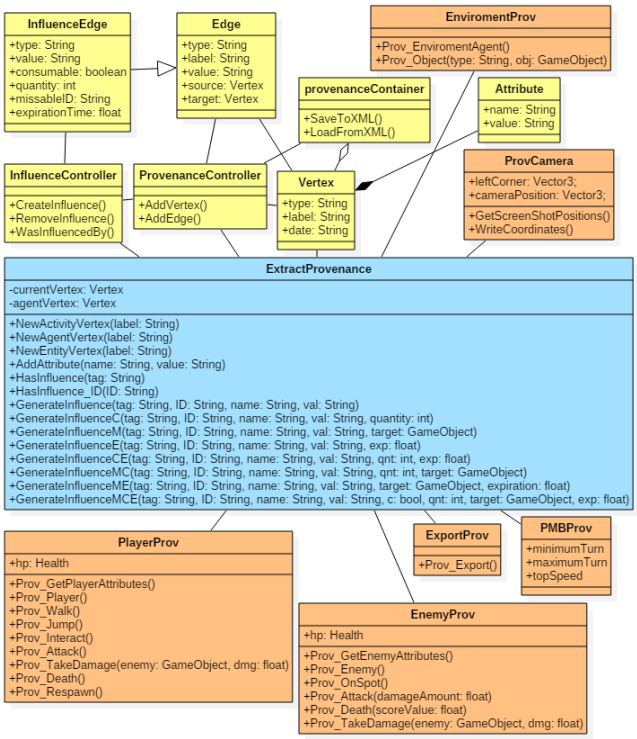

Figure 1: Simplified class diagram for PinGU.

4.1 Integrating PinGU into an Existing Game

In order to capture provenance data from a game, a game developer can use PinGU, which is available at GitHub[2]. We use the game *2D Platformer Tutorial*[3] from Unity as a running example of the PinGU integration. Figure 2 shows a screenshot of the game where the player has to kill aliens to gain score points. The game has two different types of enemies and the player can collect two different types of items to aid in his fight (health and ammunition items).

The first stage of usage consists on creating a game object in the scene to act as a centralizing server for the provenance information. This game object will have two attached classes: *ProvenanceController* and *InfluenceController*, which is illustrated in Figure 3. As said earlier, both classes are used to manage all provenance information and graph generation, thus only one instance of each are necessary per game scene. If the game is comprised of multiple scenes, then each scene will have its own provenance graph. These two classes use the other *Core* classes, which act as libraries and must not to be place in the scene.

[2] http://gems-uff.github.io/ping/

Figure 2: 2D Platformer game.

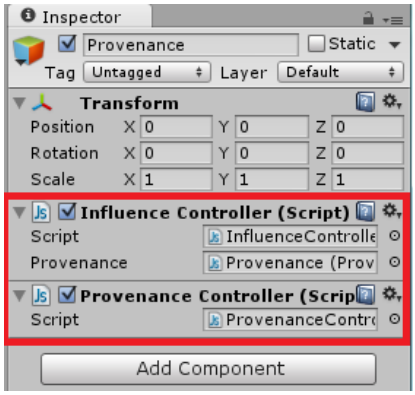

Figure 3: 1st stage for PinG integration, showing the Provenance game object and its scripts.

The second stage is to attach the *ExtractProvenance* class in each character or entity in the game (i.e. NPCs, player, interactive objects, prefabs) and link it to the object created in the first step. This class is responsible for creating all the provenance vertices for the game entity that is attached to and then passing these vertices to the *ProvenanceController* to insert it in the graph. Figure 4 illustrates an example of adding the class to the *Hero* game object, which is the player's avatar from the 2D Platformer.

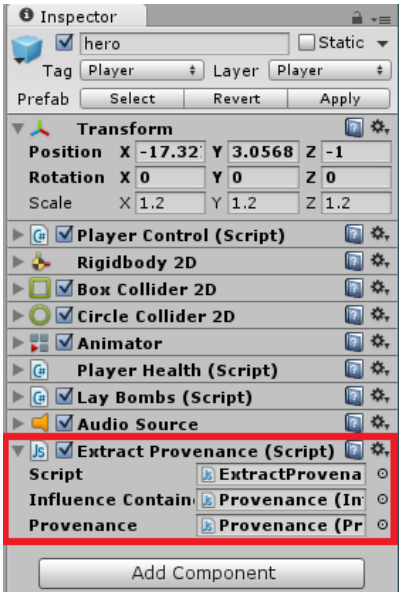

Figure 4: 2nd stage for PinG integration, showing the insertion of the provenance tracking class in existing agents and entities.

[3] https://www.assetstore.unity3d.com/en/#!/content/11228

The third stage is to identify the actions and their interactions with other actions in the game design document. In the running example, we identify the existing classes that contain the actions that we want to track, which is illustrated by Figure 5. The same figure also shows a summary of each selected class and their responsibility in the game, grouped by the identified agents (*i.e.*, *Enemy*, *PlayerControl*, *PickupSpawner*). The classes for the agents also contain additional actions, such as spawning item and movement.

The fourth stage is creating the domain-specific *provenance tracking functions* and attaching it to each entity in the game that has the *ExtractProvenance* module. Each existing module should have a *provenance function* for each possible action that the entity can perform and that we are interested in tracking.

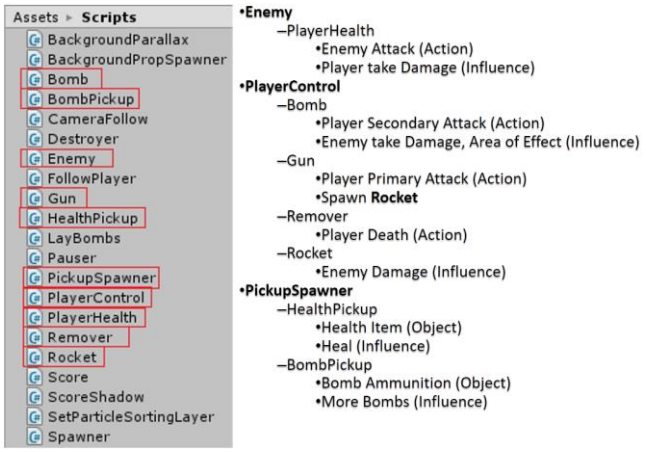

Figure 5: 3rd stage for PinG integration, showing the 2D Platformer classes and Game Design.

Unfortunately, it is necessary to create these provenance function calls due to domain contextual information. However, all these *provenance functions* are small and simple, following the same four-step recipe and changing only the context information used during each step:

1. Add game-related attributes (*e.g.,* health points, experience points, etc.);
2. Create the appropriate vertex (Activity, Agent, or Entity);
3. Check for influences (if applicable);
4. Generate influence (if applicable).

The first step is used to configure the desired information to be extracted during the execution of each action or event. They will appear at the graph's vertices as attributes. Unity already provides default attributes, such as location, tag, object name. However, game-sensitive attributes such as health points, magic points, and player score must be manually added by the *AddAttribute*(<name>, <value>) function of *ExtractProvenance* class. After adding the desired attributes, the second step creates the provenance vertex and places it in the graph. This vertex can be any of the three provenance types and must be specified by the user by calling the *NewActivityVertex*, *NewAgentVertex*, or *NewEntityVertex* functions.

The third and fourth steps are related to influence. The third step is used to verify if there is any influence that can affect the current action. If so, they are automatically inserted in the graph as an edge connecting the respective vertices. This verification can be made by a tag (*HasInfluence(<tag>)*), which is used to group a collection of influences that has something in common, or by an influence ID (*HasInfluence_ID*(<ID>)).

The forth step is responsible of creating influences (*GenerateInfluence*), so they can be used by the third step. Influences can be created with some restrictions: They can expire when a certain time passes (*e.g.*, spell duration), leading to the E (expire) suffix at the function (*i.e.*, *GenerateInfluenceE*), or after a number of times used (*e.g.*, spell that block the next X attacks) leading to the C (consumable) suffix (*i.e.*, *GenerateInfluenceC*), or both (*GenerateInfluenceCE*). There is another type of influence that can be combined with the restrictions above, which represents something that was expected to happen but for some reason it did not. For example, there is a health item in the scene that the player is supposed to get, but he forgot or skipped it. Thus, if the player did not get it, then an influence is generated saying that the player "missed" the item. However, if the player did in fact get the item, then the normal influence (effect of getting the item) occurs. For those, the function has the suffix M ("missable") (*i.e.*, *GenerateInfluenceMC*, *GenerateInfluenceMCE*).

Code 1 shows an example of a provenance function for our running example of one of the possible actions that can be executed by an enemy. The calls used in the *Prov_Attack* are implemented in the *ExtractProvenance* (*NewActivityVertex*, *HasInfluence*, *GenerateInfluenceCE*), with the exception of *Prov_GetEnemyAttributes*, which is domain related and the developer need to specify the desired attributes for tracking, besides the default attributes from Unity (*i.e.*, Tag, object name, object coordinates). This is accomplished by creating a function (e.g., *Prov_GetEnemyAttributes* from the auxiliary classes) that invokes the function *AddAttribute* from *ExtractProvenance* by passing the attribute name and value for each attribute, as illustrated by Code 2.

```
public string Prov_Attack(float damageAmount)
{
    Prov_GetEnemyAttributes();
    prov.NewActivityVertex("Attacking","");
    prov.GenerateInfluence("Player", this.GetInstanceID().ToString(),
    "Damage", (-damageAmount).ToString());
    return this.GetInstanceID().ToString();
}
```

Code 1: PinG code for tracking game data. Orange text in the code is domain-related.

```
// Enemy Attributes
public function Prov_GetEnemyAttributes()
{
    prov.AddAttribute("Health", hp.health.ToString());
}
```

Code 2: Example of a provenance function for tracking attributes.

After creating the necessary *provenance functions* for their respective game objects, the next step is to incorporate the function calls in existing game classes in order to register the provenance information. All this process becomes trivial if the developers have a detailed game design document stating all the possible actions that can be executed in the game along with their purpose. The action list shows the actions that are desired to be tracked and the necessary provenance functions that need to be made. Meanwhile, the action's purpose gives us insights on the influences that they can generate during or after executing the action.

Code 3 shows an example of code insertion in an existing game module responsible for controlling the artificial intelligence (AI) of enemy characters in the game. The "damageAmout" is a configurable variable from the original class that states the damage the attack will cause. We inserted the provenance call for the *Prov_Attack* function, whose code appears in Code 1 in the function responsible to make the enemy AI fire at the player. We added a

package of auxiliary classes that, depending on the type of the game, does the majority of the work and requires only coding the function call in the existing game classes. Furthermore, they can also be used as a guiding example in cases that the desired action is not already implemented. These classes are *PBMProv*, *PlayerProv*, *EnemyProv*, and *EnviromentProv*, and each is customized for the particular type they represent (Car-related movements, Player, Enemy, and Environment).

```
function Fire () {
    if (weaponBehaviours[nextWeaponToFire]) {
        weaponBehaviours[nextWeaponToFire].SendMessage ("Fire");
        nextWeaponToFire = (nextWeaponToFire + 1) % weaponBehavi
        lastFireTime = Time.time;

        // Provenance
        prov.Prov_Attack(damageAmount);
    }
}
```

Code 3: Provenance function call insertion into existing classes.

The last step is to add a provenance export function to an event so it can save the current provenance graph to an external xml file when the designated event is executed (*e.g.*, player's death, completing the level). Code 4 illustrates the *provenance functions* for our running example responsible for exporting the tracked data, which is linked to the player's death, and Code 5 shows the insertion of the *provenance function* call to track the information.

```
public void Prov_Death()
{
    Score score = GameObject.Find("Score").GetComponent<Score>();

    prov.AddAttribute("Health", "0");
    prov.AddAttribute("Score", score.score.ToString());
    prov.NewActivityVertex("Death","Drowned");
    Prov_Export();
}
void Prov_Export()
{
    Debug.Log ("Exported");
    GameObject ProvObj = GameObject.Find("Provenance");
    ProvenanceController prov = ProvObj.GetComponent<ProvenanceController>();
    prov.Save("2D_Provenance");
}
```

Code 4: Provenance function for the player's death action.

The PinGU integration is explained with more detail in the tutorial available at the component's GitHub page, showing all provenance functions and their insertion in the identified modules. Figure 6 shows an example of the generated provenance graph from the tracked actions executed during a game session, which was rendered using *Prov Viewer*. We can see in this graph the player's and each enemies' actions and how they interacted with each other by looking at the vertical colored edges.

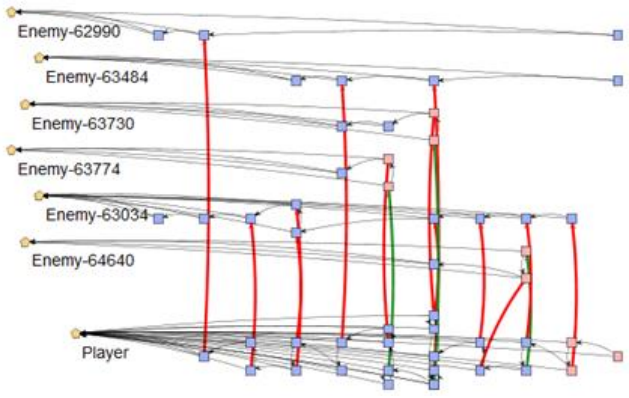

Figure 6: Example of the generated graph for the 2D Platformer.

```
void OnTriggerEnter2D(Collider2D col)
{
    // If the player hits the trigger...
    if(col.gameObject.tag == "Player")
    {
        // .. stop the camera tracking the player
        GameObject.FindGameObjectWithTag("MainCamera").GetComponent<CameraF

        // .. stop the Health Bar following the player
        if(GameObject.FindGameObjectWithTag("HealthBar").activeSelf)
        {
            GameObject.FindGameObjectWithTag("HealthBar").SetActive(false);
        }

        // ... instantiate the splash where the player falls in.
        Instantiate(splash, col.transform.position, transform.rotation);
        // ... destroy the player.
        Destroy (col.gameObject);

        //Provenance Death
        Prov_Death(col.gameObject);

        // ... reload the level.
        StartCoroutine("ReloadGame");
    }
    else
    {
        // ... instantiate the splash where the enemy falls in.
        Instantiate(splash, col.transform.position, transform.rotation);

        // Destroy the enemy.
        Destroy (col.gameObject);
    }
}
```

Code 5: Fragment of the original Remover module: Added the provenance function call in the player's death.

4.2 Capturing Game Scene

We also implemented a specialized camera module in order to simplify the process of capturing the game map to use it in combination with the provenance graph. This camera is orthographic, which preserves the dimensions and does not change coordinates to accommodate the perspective of the viewer. Thus, this camera needs to be placed either directly above the game scene or laterally (for platform games), allowing it to capture the entire map. This module automatically captures the screenshot of the scene and the necessary data required to align the provenance graph, which uses world space coordinates, with the captured map, which uses pixel position. The screenshot resolution can also be adjusted in the module.

The camera module captures the camera's world position (*cameraPosition*) and the camera's upper left corner coordinates in world position (*leftCorner*). The camera's position is used to translate the game map in order to align it with the graph and is easily obtained by getting the position of the camera in world space. The second information is used to scale the graph to match the picture and is captured by converting the camera position from viewport space to world space, which is the upper left corner.

In order to align the graph with the map, it is necessary to find a scale factor, that can be trivially be calculated by Equation 1.

$$scaleFactor = \frac{0.5 \times pictureWidth}{leftCorner.x - cameraPosition.x} \quad (1)$$

The *scaleFactor* is used to transform the world coordinates captured from the provenance data to pixel coordinates used in the screenshot of the game map. Therefore, the game designer only needs to position the orthographic camera in the game scene and add the camera module in order to capture the entire map and the necessary data. After that, the designer can use the coordinates captured by the module and the screenshot in a visualization tool.

4.3 Provenance Graph Visualization

One of the purposes of collecting provenance data is to be able to generate a provenance graph to aid the developer in analyzing and

inferring the reasons of the outcomes. After incorporating the PinGU approach into an existing game, the provenance data is captured and stored while a game session is being played. Afterwards, users can generate a provenance graph for that specific game session.

The generated provenance graph is exported to a simple XML file containing a list of vertices and edges in the graph. This data can be used for data mining, exploration, and visualization. For this work, we employ an open-source provenance visualization tool named *Prov Viewer*[4] [18], which uses a graph framework to allow detailed rendering and visual data analysis and exploration of the provenance information. The tool provides many visualization and manipulation features: (1) collapsing, highlighting the relevant information in the graph; (2) filtering, removing information that is not relevant for a given analysis; (3) graph merge, integrating the analysis of multiple game sessions; (4) specialized layouts, organizing the graph in a more understandable way; (5) domain configuration, customizing the visualization for specific needs; and (6) shapes, sizes, and colors, supporting a clear distinction of information types. Figure 7 illustrates the tool's architecture, highlighting its main features.

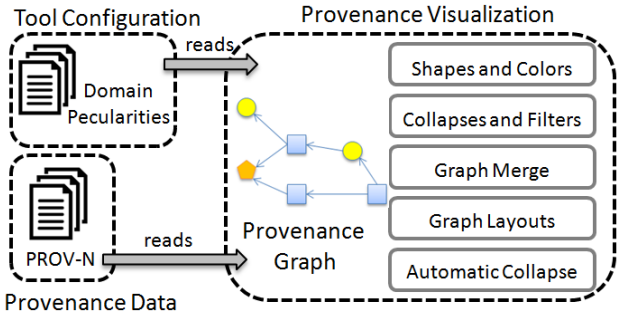

Figure 7: Prov Viewer's high-level architecture (from [18]).

When evaluating tracked attributes, *Prov Viewer* uses traffic light scheme to quickly differentiate values, thus changing vertex color to the appropriate shade. The shades vary from red to green, with yellow as the middle term. Similarly, edges also use shades to distinguish values of the same type (*e.g.*, damage), as well as thickness to show how strong the relationship is. Bright red represents negative values, bright green represents positive values, and darker shades represent values near zero. This feature allows the user to quickly identify strong influences in the graph just by looking at the edge's thickness and their color. Figure 8 illustrates some of these visualizations features in action.

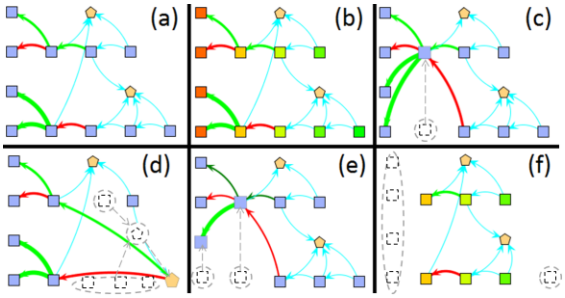

Figure 8: (***a***) Original graph; (***b***) graph with a color schema; (***c***) collapse of two activities; (***d***) collapsing of the agent's activities; (***e***) graph c after another collapse; and (***f***) temporal filter. (from [18]).

The tool also has a spatial layout that organizes the vertices in the graph by their spatial coordinates and can be used for spatial or geo-referencing the data. The layout supports the usage of an orthographic image, which is captured in the PinGU component. This is particularly useful for corresponding elements with other graphical representations, such as a map of the game scene. When using the spatial layout in conjunction with a background image, the user can see where each tracked event occurred just by looking at the graph's placement in the image. All the graph images in the following sections were rendered using *Prov Viewer*.

5 Case Study

The following sub-sections present two open-source game samples (*Car Tutorial*[5] and *Angry Bots*[6]) where we demonstrate the generated provenance graphs by incorporating PinGU. In the first game, we focus on showing that the provenance data can facilitate the graph analysis on how previous actions or events affect future actions. We also show how the provenance graph evolves when the game has multiple cycles. In the second game, we show another case of provenance data with a different genre of game, allowing for easy identification of sections that were not explored by the player and where he/she had more difficulty. We did not modify the games in any way nor added new features besides coupling with the PinGU, which is only responsible for tracking provenance data. Both case studies use *Prov Viewer* tool for visualizing the provenance graphs.

5.1 Car Tutorial

The first case study is the Car Tutorial from Unity asset store. This tutorial has only one racetrack and focuses on the arcade style racing game. In addition, there is no implemented AI for opponent cars. Following the conceptual framework, PinG tracks events and actions executed during the game session, along with their effects on other events, to compose the provenance graph (*e.g.*, crashing the car, pressing the car's brake).

We can use the car's coordinates in the track to plot the graph so that it is possible to visualize where the player was when the action was executed. This visualization also allows the designer to quickly identify which sections of the track the player had trouble. Thus, we can take advantage of spatial-referencing the data during the provenance visualization. We used a screenshot of the game map taken by our camera module with dimensions of 1070x802.

Figure 9 shows the provenance graph of one game session, using the car's coordinates and the track's picture as background. This graph is composed of 169 vertices and 867 edges extracted from a 107-second game session, which represents one complete lap in the track. The vertices are colored according to the car's speed (gradient from white when close to zero and green for high values) and the visible edges are the speed delta between vertices.

We can quickly identify sections of the track that the player may have had issues, either by reducing the speed too much or by crashing, by just looking at the plotted graph in the race track. As an example, Figure 10 shows a zoomed section of the graph to better illustrate the reasons behind a car crash. The zoomed section of the graph has a different vertex-coloring scheme to differentiate events. By analyzing it, we can see that the car crash (red vertex) was influenced by two factors. The first one was on the previous curve, where the car lost contact with the ground (purple vertex with a blue edge linking the crash) after passing through the rumble strips at the end of the maneuver, thus preventing the player to prepare for the following turn. The second reason was that the player was too fast, as indicated by the red edge from the blue vertex, which is a reduction of the car's turn rate due to high speed.

[4] http://gems-uff.github.io/prov-viewer/
[5] https://www.assetstore.unity3d.com/en/#!/content/10
[6] https://www.assetstore.unity3d.com/en/#!/content/12175

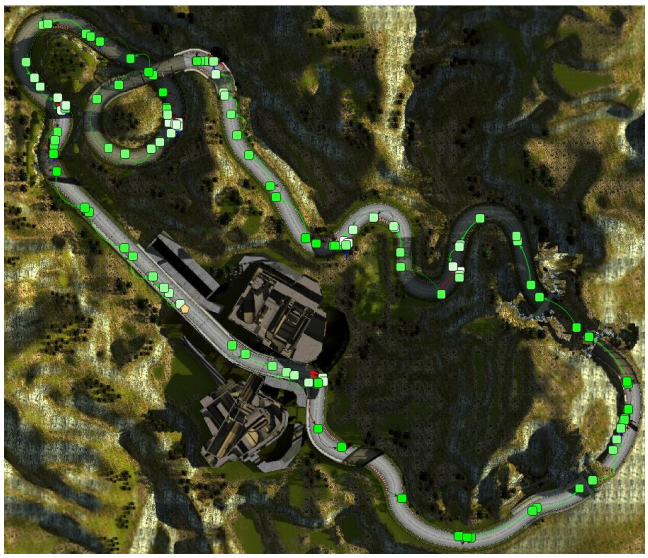

Figure 9: Spatial referencing provenance data.

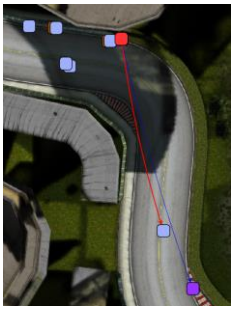

Figure 10: Influences behind a car crash.

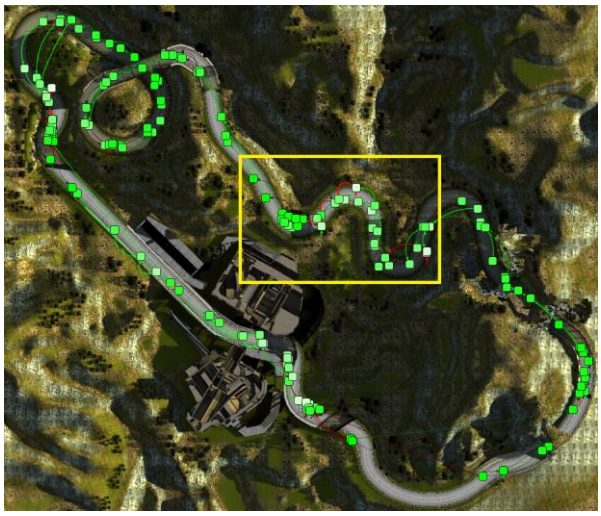

Figure 11: Provenance graph from multiple laps.

Using the tracked telemetry data from other laps of the race, we can begin to detect patterns during the game session or even compare the player's performance between laps. This analysis can also be extended to different game sessions by comparing the generated provenance graphs. Figure 11 illustrates an example of the generated provenance graph when gathering data from multiple laps during a single play session, enabling the designer to detect behavioral patterns and locations where the players are struggling the most. For example, Figure 12 shows a section of the track that is characterized by having multiple curves in the track. We can see the player's performance during each lap of the race, where each lap is represented by a different edge color. The first, second, and third laps are presented by red, green, and blue edges respectively. Moreover, the first and last vertices of each lap are marked with circles of the same color as the edge and the timestamps are represented by the yellow numbers beside the vertex. As we can see, the player had approximately the same speed in all laps due to having the same shade of green when entering this section of the track. However, the player took fifteen seconds to pass through this section of the track on his first lap (52 - 37), seventeen seconds during the second lap (131 – 114), and ten seconds on the third lap (200 – 190).

By analyzing Figure 12, we can see a purple edge that represented the reason behind the crash in the first lap (marked by the purple circle). This purple edge represents a cause-and-effect relationship, showing that the crash happened because the player passed through rumble strips (brown circle) and, as a result lost car stability, could not complete the turn. Furthermore, notice the steep angles the player had to make due to his positioning in each curve. During the second lap (green edges), the player tried to avoid the crash by reducing speed. However, the player reduced too much speed to enter the second curve (white-green vertices). During the third lap (blue edges), the player managed to improve his performance and avoid any crashes by better positioning the car before each curve and thus reducing the necessary angle to make the turn while maintaining a nearly constant speed.

Figure 12: Zoomed section from yellow rectangle on Figure 11.

5.2 Angry Bots

We conducted a second case study using a very different style of game, called Angry Bots, also from the Unity asset store. Angry Bots belongs to the hack-and-slash genre, being a top-down action shooter. In the available scenario, the player has to face enemy robots and interact with the environment in order to complete the level. Figure 13 illustrates one of the possible visualizations of the provenance data gathered by our component, showing the vertex visualization scheme for the player's health attribute value (vertex color using a traffic light scheme) and the edges that influences in it (green and red edges) as the game progresses. Blue vertices represent other characters in the game (enemies), blue edges represent the chronological order of events, and green edges represent player's health generation due to his passive regeneration ability. By analyzing Figure 13, we can see the chronology of events, regions visited by the player, sections where more action happened, places where the player engaged in battle, and when the player suffered heavy health loss. In this game, we used a screenshot taken by our camera with the dimensions of 4280x3208 in order to show that the figure size does not affect the graph alignment process. This increase of resolution allows for a higher detail of the game scene visualization when zooming the graph during analysis.

Figure 13: Picture of the entire graph. Vertex coloring based on player's Health attribute.

Considering that the player recovers health periodically, it is possible to infer that the cause of some deaths was the rush through the level without waiting to recover health or because of a tough enemy. Figure 14 illustrates the first case, where the player tried to rush through the game without waiting to regenerate the player's health, lost from the previous battles. The light blue arrows were added in the figure to highlight the player's general movement and does not belong to the provenance data.

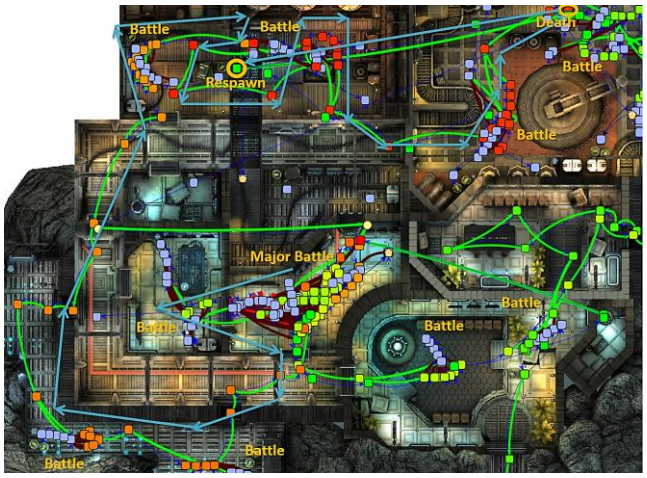

Figure 14: Player's health when trying to rush the game.

After the player engaged an enemy in a major battle, which the player didn't leave unscathed by looking at the orange vertices, the player continued advancing through the level. Then, on the player's third major engagement, where he was still wounded by looking at the orange vertices, the player lost the majority of his remaining health, as illustrated by the following red vertices. Even though the player was low on health, he managed to dispatch his enemies on the forth battle without losing a single health point (no red edges). However, the player continued pressing on without resting, which would allow for him to gradually restore his lost health points before his next engagement, until dying on the next battle when the player got hit by the enemy (Battle #5).

Figure 15, Figure 16, and Figure 17 illustrates the second case, showing the sequence of events that led the player to a tough engagement (Figure 17). By analyzing the picture, we can see that the player started these events (Figure 15) with good health (green vertices), leaving the first battle slightly injured (yellow vertex). A few moments later he encountered another enemy in a side room (Figure 16), where once again he overcame the enemy with only minor wounds (vertex is still yellow). However, just when he left

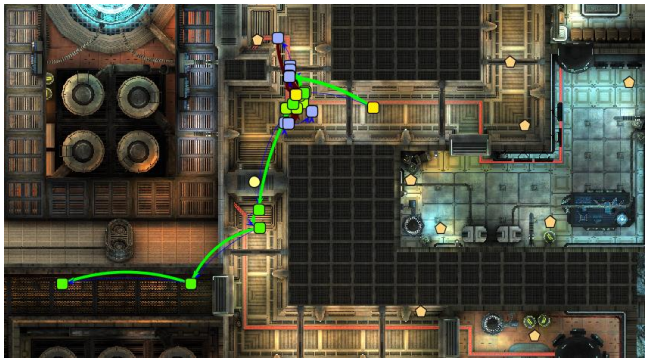

Figure 15: Sequence of events of the player exploring a section of the map and confronting an enemy.

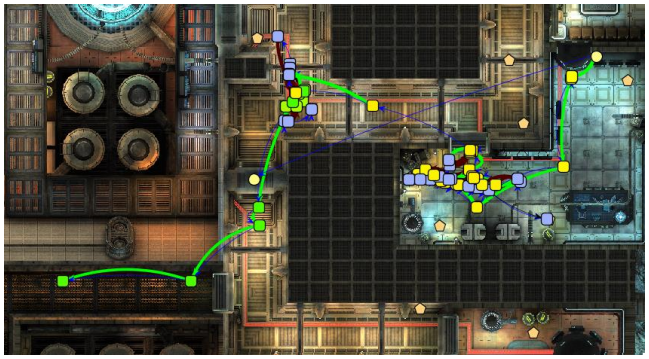

Figure 16: Continuation of the sequence of events from Figure 15 with a second engagement inside a room.

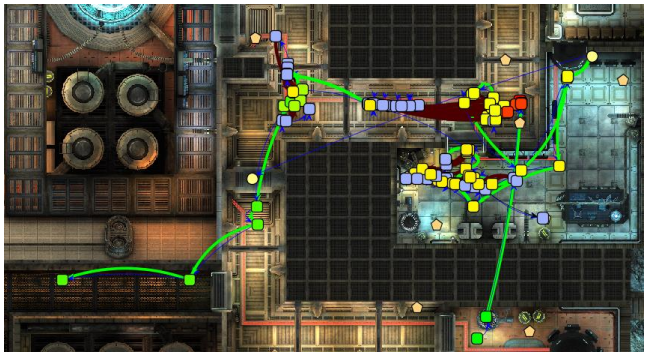

Figure 17: Continuation of the events from Figure 16 that led the player to a tough confrontation that resulted in his death.

the room, the player was ambushed by another enemy that was patrolling the corridor (the new blue vertices in the corridor from Figure 17). This enemy was a mech, which is much tougher than a regular enemy (notice the high number of dark red edges that represent player doing damage to the enemy). This battle resulted in the player's death after getting hit by two rockets (Figure 18) followed by his resurrection shortly after (green vertex in the bottom of Figure 17 that is linking the green edge to a red vertex).

Figure 18: A zoomed section from Figure 17 showing both the moments the player was hit by the enemy's rockets. Filtered to show only the edges that affected the player's Health.

Figure 19 illustrates the moments when the player died, which are marked by red circles. Meanwhile, the orange circles illustrate the player "refreshed" state after resurrecting, as well as the resurrected location. Both situations have a green edge linking the player's death to the resurrection, which shows that his health went from zero (red vertex) to maximum (green vertex) after resurrecting. Notice that the player actually died three times trying to beat the enemy mech from Figure 17 before finally defeating it.

Figure 19: Filtered graph showing the moments the Player died and was resurrected.

6 Conclusion

This paper presented the concretization of PinG, a conceptual framework for game telemetry that tracks the actions and events alongside with their cause-and-effect relationships, through a component in Unity. Our component facilitates the process of tracking and storing the provenance knowledge for data exploration and analysis. This provenance knowledge can aid the detection of gameplay issues, support developers for a better gameplay design, identification of game sections where players had issues and the reasons behind these issues, and mining behavioral patterns from individual sessions or groups of sessions.

Moreover, we showed two games that used our PinG component to extract provenance knowledge, giving examples of analyses from the provenance data. These examples demonstrated the possibility of referencing the provenance knowledge in the game map to better visualize and understand the events of a game session. Despite not showing the typical, and simpler, existing game data analytics techniques and data mining, we believe that the richness of the provenance data extracted when using the PinG approach and our component provides the necessary means to make possible deeper game data analyses.

We are currently working on ways to improve the PinG component to automate even further the data tracking, especially for influences and, in the future, possibly implement the component for other engines, such as Unreal Engine due to their recent business change for indie developers. Moreover, due to the quantity of the data extracted with PinG, we are studying techniques to improve even further the visual analysis process. These studies involve, but are not limited to, automatic graph inferences, data mining, graph reduction, multiple graph analysis to compare multiple game sessions or even cycles during a game (*e.g.*, laps in a racing game), enabling better strategies of provenance gathering that take advantage of the game's genre and type.

References

[1] M. El-Nasr, A. Drachen, and A. Canossa, Eds., *Game Analytics - Maximizing the Value of Player Data*. In: Springer Science & Business Media, 2013.

[2] G. Zoeller, "Development telemetry in video games projects," *Game Dev. Conf. GDC*, 2010.

[3] R. Hunicke, "The Case for Dynamic Difficulty Adjustment in Games," in *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*, New York, NY, USA, 2005, pp. 429–433.

[4] A. Drachen, R. Sifa, C. Bauckhage, and C. Thurau, "Guns, swords and data: Clustering of player behavior in computer games in the wild," *Conf. Comput. Intell. Games CIG*, pp. 163–170, 2012.

[5] C. Pedersen, J. Togelius, and G. N. Yannakakis, "Modeling Player Experience for Content Creation," *Trans. Comput. Intell. AI Games T-CIAIG*, vol. 2, no. 1, pp. 54–67, Mar. 2010.

[6] B. G. Weber, M. John, M. Mateas, and A. Jhala, "Modeling Player Retention in Madden NFL 11," *Innov. Appl. Artif. Intell. Conf. IAAI*, 2011.

[7] G. Wallner, "Play-Graph: A Methodology and Visualization Approach for the Analysis of Gameplay Data," *Found. Digit. Games FDG*, pp. 253–260, 2013.

[8] Y.-E. Liu, E. Andersen, R. Snider, S. Cooper, and Z. Popović, "Feature-based projections for effective playtrace analysis," *Found. Digit. Games FDG*, pp. 69–76, 2011.

[9] M. S. El-Nasr and T.-H. Nguyen, "Glyph: Visualization Tool for Understanding Problem Solving Strategies in Puzzle Games."

[10] S. Joslin, R. Brown, and P. Drennan, "The gameplay visualization manifesto: a framework for logging and visualization of online gameplay data," *Comput Entertain*, vol. 5, no. 3, p. 6, 2007.

[11] J. H. Kim, D. V. Gunn, E. Schuh, B. Phillips, R. J. Pagulayan, and D. Wixon, "Tracking real-time user experience (TRUE): a comprehensive instrumentation solution for complex systems," *Hum. Factors Comput. Syst. CHI*, pp. 443–452, 2008.

[12] T. Kohwalter, E. Clua, and L. Murta, "Provenance in Games," *Braz. Symp. Games Digit. Entertain. SBGAMES*, pp. 162–171, 2012.

[13] PREMIS Working Group, "Data Dictionary for Preservation Metadata," Implementation Strategies (PREMIS), OCLC Online Computer Library Center & Research Libraries Group, Final report, 2005.

[14] T. Kohwalter, E. Clua, and L. Murta, “SDM – An Educational Game for Software Engineering,” *Braz. Symp. Games Digit. Entertain. SBGAMES*, pp. 222–231, 2011.

[15] T. Kohwalter, E. Clua, and L. Murta, “Game Flux Analysis with Provenance,” *Adv. Comput. Entertain. ACE*, pp. 320–331, 2013.

[16] L. Jacob, T. Kohwalter, E. Clua, D. De Oliveira, and A. Machado, “A Non-intrusive Approach for 2D Platform Game Design Analysis Based on Provenance Data Extracted from Game Streaming,” *2014 Braz. Symp. Comput. Games Digit. Entertain. SBGAMES*, pp. 41–50, Nov. 2014.

[17] T. Kohwalter, E. Clua, and L. Murta, “Reinforcing Software Engineering Learning through Provenance,” *2014 Braz. Symp. Softw. Eng. SBES*, pp. 131–140, Sep. 2014.

[18] T. Kohwalter, T. Oliveira, J. Freire, E. Clua, and L. Murta, “Prov Viewer: A Graph-Based Visualization Tool for Interactive Exploration of Provenance Data,” in *Proceedings of the 6th International Provenance and Annotation Workshop on Provenance and Annotation of Data and Processes - Volume 9672*, New York, NY, USA, 2016, pp. 71–82.

[19] Y. Gil and S. Miles, “PROV Model Primer,” 2010. [Online]. Available: http://www.w3.org/TR/prov-primer/. [Accessed: 21-Mar-2013].