International Journal of Multidisciplinary Research and Publications
ISSN (Online): 2581-6187

Implementation Simple Additive Weighting in Procedural Content Generation Strategy Game

Samuel Putra[1], Wirawan Istiono[2]
[1,2]Informatic, Universitas Multimedia Nusantara, Tangerang, Banten, Indonesia
Email address: samuel4@student.umn.ac.id, wirawan.istiono@umn.ac.id

Abstract**—** *Strategy Games allows players to play a never-ending puzzle with another person or computer, unfortunately computers has a static thinking flow causing the human player to be able to recognize patterns from computer opponents. Single player strategy games uses large amounts of game contents to deal with this problem, but today game content is becoming more and more expensive to create. Procedural Content Generation has recently been a popular method to solve this problem because of it's potential to create an endless amount of generated content. However a PCG needs many rules in order to make a playable and appealing content. This research creates a PCG that with an implementation using a Simple Additive Weighting Method which is used to calculate player inputs into a single value and then distribute that single value into multiple outputs created by the PCG. After the game has been created user experience was than measured before and after the SAW implementation was added using Game Experience Questionnaire. After the SAW implementation players positive effects towards the game increased by 23.5%, negative effects towards the game decreased by 29.83%, players competence increased by 19.83%, and the games overall challenge decreased by 22%.*

Keywords**—** *Game Experience Questionnaire, Procedural Content Generation, Simple Additive Weighting, Strategy Games*.

I. INTRODUCTION

Strategy games, or especially those discussed in this study, turn-based strategy games have a history that can be found in civilizations thousands of years ago in places like Rome, Greece, Egypt, and so on. The strategy game found from various places and times that still has relevance today is chess. Chess has simple rules but has a complexity that is still being explored by various players around the world because it has unlimited strategy possibilities. Strategy games give players time to plan their actions without taking into account the player's reactions [1]. Turn-based games also have the advantage of enabling games on a larger scale without burdening players with too much information [2].

When compared to strategy games made today, for example the Fire Emblem game series or XCOM, it is necessary to consider games played alone against the computer, so these games rely on a lot of different content to create variety in the game. Michael Toy created Rogue in 1980 with the aim of creating a strategy game played by the player and the game world itself. Rogue is designed to change every time it is played so that players cannot win the game just by memorizing the layout design in each level of the game [3], this concept inspires a new genre in game development, namely roguelike. Now games with roguelike elements are popularly used by independent games such as Spelunky (2009), The Binding of Isaac (2011), and Hades (2020) which are also known as hybrid roguelikes. The gameplay elements that define roguelikes are often debated, but what is universally agreed upon is the element of randomization in game content [4], usually using procedural content generation and permadeath methods which means the character used by the player cannot be used again and must start from the beginning of the game again. Procedural Content Generation (PCG) allows games to always create unique obstacles for players, thus creating a new user experience for each game [5].

Content created by PCG is usually of lower quality than if it was created by human designers [6]. PCG is also more difficult to control from a usability and visual perspective. For example if using the previous example there is a possibility that the level made is impossible to complete because there is no way out, a room is blocked, or some other problem occurred [7]. The levels created also have the possibility of looking messy, making the content recognizable as made by computer algorithms [4], [8], and not by humans, thereby reducing the user experience. PCG needs a lot of additional features to prevent these problems from happening, but the more features added the more complicated the game system mechanics are developed and the harder it is to iterate [9].

Based on the problems described, this research will design and build a PCG system that implements Simple Additive Weighting to account for the generated content. PCG in this study takes into account the level of difficulty and complexity in the game that changes as the game progresses, so SAW is suitable to be used because the method is quite simple and flexible to be used in various situations [10], [11].

II. METHODOLOGY

The research methodology that will be used for the design and development of strategy games using Procedural Content Generation with the implementation of the Simple Additive Weighting method, the first step in this research is Literature Study, where in the literature study, research was conducted on theories related to Game Design, Game Progression, Level Design, Simple Additive Weighting Method, PCG, and how to evaluate PCG.

Second step, in this research is creating the game, where the game is made using the Unity software and using the C# programming language. The game mechanism is made based on the information obtained from the literature review. The first stage of game creation is the creation of the game framework, this includes the creation of game logic from the

Samuel Putra and Wirawan Istiono, "Implementation Simple Additive Weighting in Procedural Content Generation Strategy Game," *International Journal of Multidisciplinary Research and Publications (IJMRAP)*, Volume 4, Issue 12, pp. 9-18, 2022.

game view, how players can interact with the game, interactions between game objects, artificial intelligence, etc. After designing the game framework, a PCG is made that can manipulate the content created during the creation of the game framework. After PCG is complete, the SAW method is implemented to improve the quality of content made by PCG. Then designed, user interface, and other additional features.

The third step is game testing, where in this step testing is done by playing and seeing the success of the mechanics of the game made. After ensuring that the game mechanism runs properly, the user experience will be tested using a respondent's questionnaire. A group of respondents will be tested by looking at the way the respondents played the game and then using a questionnaire to get feedback from the user experience obtained before and after the implementation of SAW. The final step is game evaluation, where the game evaluation is done through analyzing the results of the questionnaire. The results of the questionnaire will be evaluated using the metrics discussed in the literature review. Evaluation aims to determine whether the goals of making the game are met.

The main objective of this game is to reach the last level of the game and defeat the last enemy. The objective of each level is to reach the end point with a pawn or king piece. During the game, the player also has a secondary objective, namely to earn money by defeating enemies or taking certain objects in the level. The player enters the application and the initial view of the user is the main page of the game. In this view the user can start the game, change sound settings, or exit the game. Once the player chooses to start the game the player needs to complete the level with the given pieces at the start of the game. As long as the player plays the level the player can earn money by defeating enemies or getting certain objects in the level. After completing the first level, the next level will start with the player deciding what to do with the money earned. Players can buy additional pieces or strengthen existing pieces. Once the player is ready to start the level, the player can press the Start button and the level continues as before. If the player makes it to the last level, which is level 30 the player needs to defeat the last enemy at that level. If the player manages to defeat the enemy then the player will win the game. In the game the player can press the pause button to restart the game, or exit the game. The game will end if the players king piece has 0 health points.

Figure 1 shows the general process of creating a map that is played in a level. Based on the level points you have, the difficulty level and the size of the map will change. The map is made with a 2-dimensional integer that will be filled with values that will be populated by the game object after the PCG process is completed by the program. The process of SAW Calculate Player and SAW Distribute Points is the use of the SAW method to calculate the level of strength of the player and use these calculations to determine the difficulty in the map to be created. Random Make Walls is the process of placing walls in a map that serves as an obstacle that must be passed by the player, and Smooth Walls is the process of tidying up the walls that are made. Random Fill Gamespace is the process of placing certain enemy pieces or items into the map. Populate level is the process of using a map that is formed from all previous processes and placing game objects based on that map. If the player reaches level 30 then all these processes will be skipped and the player will play the last level that was created previously.

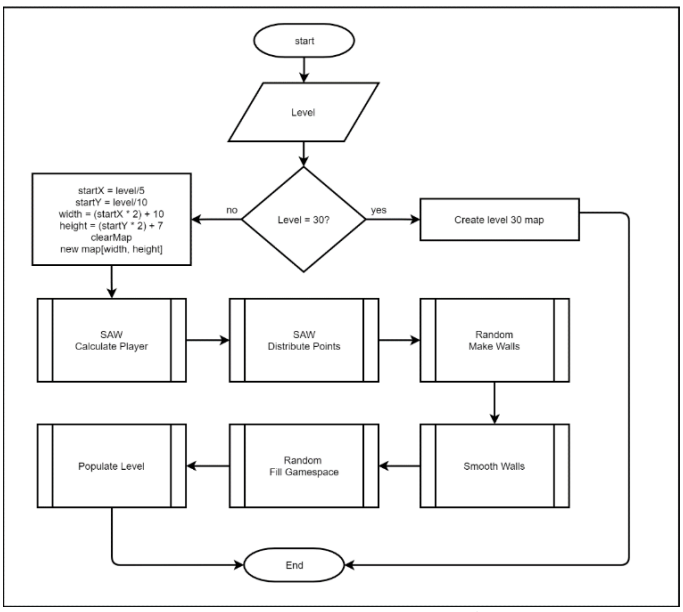

Fig. 1. Generate Map Chess Labyrinth flowchart

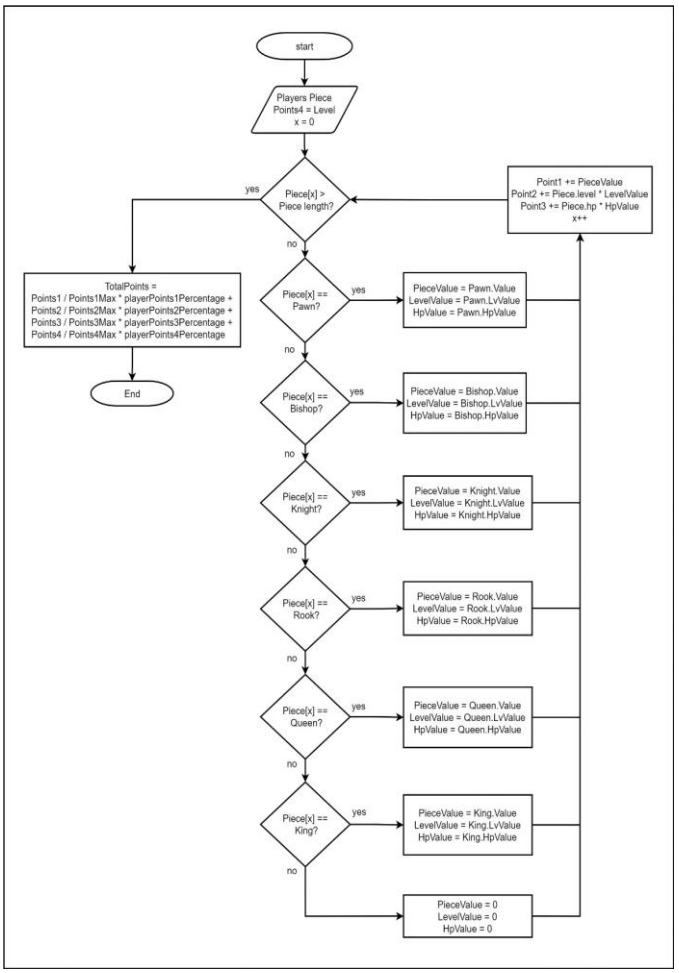

Fig. 2. SAW Calculate Player Chess Labyrinth Flowchart

Samuel Putra and Wirawan Istiono, "Implementation Simple Additive Weighting in Procedural Content Generation Strategy Game," *International Journal of Multidisciplinary Research and Publications (IJMRAP)*, Volume 4, Issue 12, pp. 9-18, 2022.

Figure 2 shows the process for calculating a player's strength level. The player's strength is measured by 4 different values, which are translated into Point1, Point2, Point3, Point4. Point1 is the number of pawns owned by the player, Point2 is the level of each pawn, Point3 is the blood point of each pawn, Point4 is the level point of the map to be created. Each piece also has a different value according to the level of use. All player pieces will be counted one by one and each point will be added according to the value of their respective importance. After each point is added up, all values are normalized and then calculated to produce a total score according to the importance of each point.

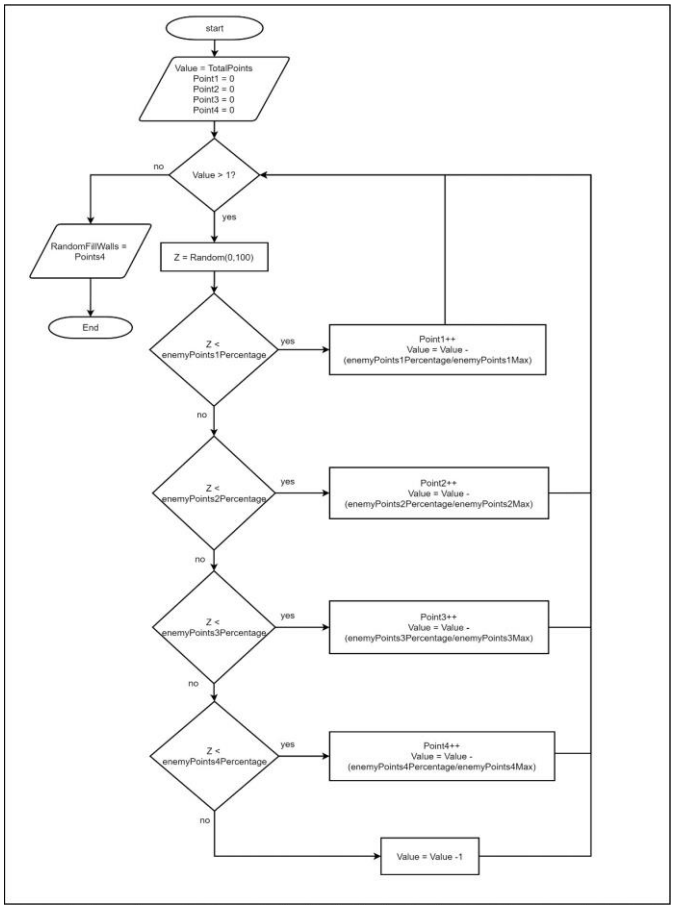

Fig. 3. SAW Distribute Points Chess Labyrinth Flowchart

Figure 3 shows the process for distributing the previously obtained values into the created map. The total previous value is distributed into several categories, namely enemy pieces, number of walls, and other game items. Categories are selected randomly and after one category is selected the total value is reduced and then the next category is randomly selected again. This process is repeated continuously until the total value is exhausted. Each category has a different amount of importance, this process uses the SAW method but in reverse. After the total value is used up RandomFillWalls is filled with the result of Point4 to assist in making the wall in the next part of the process.

Figure 4 shows the process of placing wall game objects in the level map. The number of walls is created randomly with the help of RandomFillWalls obtained from the previous process. If map[x, y] is at the edge then the value of map[x, y] will be 1. If map[x, y] is not at the end, the program will check randomly, if the random value obtained is less than RandomFillWalls then map[x, y] will be 1, otherwise it will be 0. This process will be repeated until all maps are filled with values.

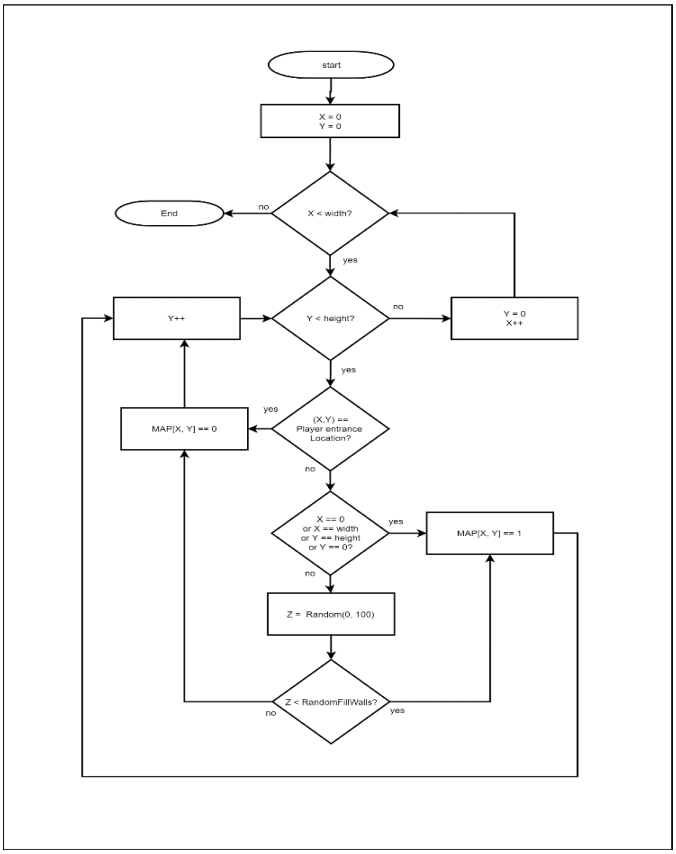

Fig. 4. Random Create Walls Chess Labyrinth Flowchart

Figure 5 shows the process for smoothing the walls placed in the previous section using the cellular automata algorithm. First the application will do a loop. Each iteration will check the neighbors of map[x, y]. if there is a neighbor with a value of 1 then the wallCount variable will increase by 1. If the total number of neighbors is more than 4 then map[x, y] will also be worth 1. If wallCount is less than 4 then map[x, y] will be worth 0.

Figure 6 shows the flow of placement of level obstacles into levels. First the number of obstacles is obtained by adding up the points 1-3 and the threshold using the total number of points obtained from the previous section, and then the variable a is used to speed up the looping process if random selection produces the same value over and over again. Then the same looping process with the placement of the walls in the previous section begins again. If map[x, y] is 0 then the program checks randomly, if the random value is more than the threshold map[x, y] is ignored and continues to the next iteration. If the random value is less than the threshold, an obstacle object will be assigned to that point.

Samuel Putra and Wirawan Istiono, "Implementation Simple Additive Weighting in Procedural Content Generation Strategy Game," *International Journal of Multidisciplinary Research and Publications (IJMRAP)*, Volume 4, Issue 12, pp. 9-18, 2022.

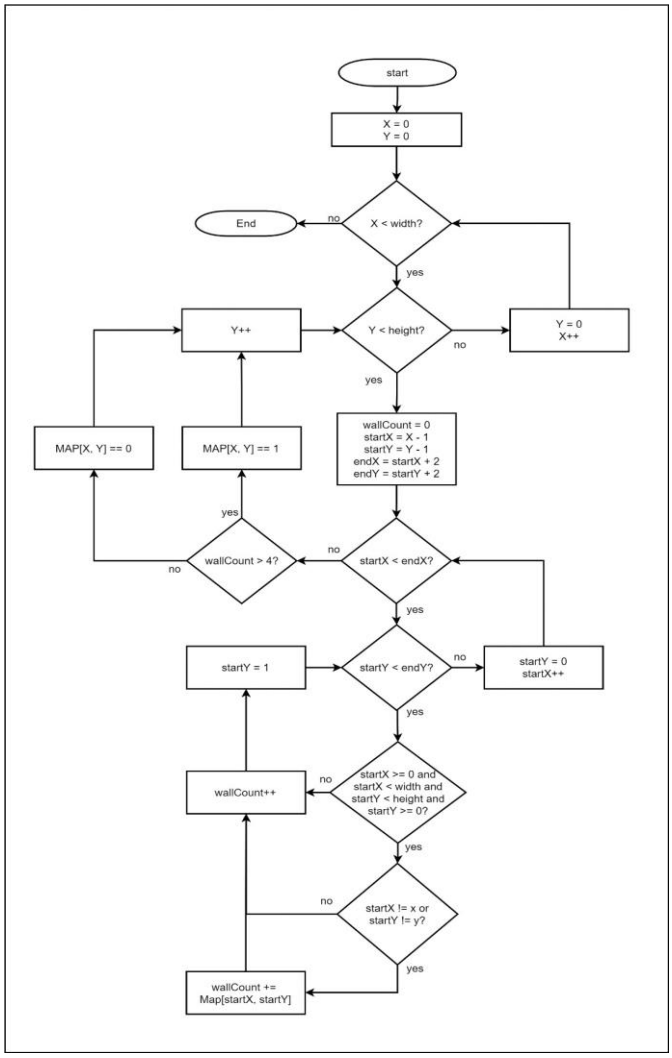

Fig. 5. SmoothWalls Chess Labyrinth Flowchart

Then the obstacle is chosen randomly, if the previously defined obstacle is selected then map[x, y] will be filled with a new value based on the corresponding obstacle value. If the selected hurdle is empty then random selection is started again until a non-blank value is obtained. When the loop is complete but not all obstacles are placed, the loop will start again from the beginning with the threshold added by the value a to add the probability of the obstacle selected by a random program.

Figure 7 shows the process of placing game objects on the map. The map required by the game has been created, now all game objects are placed in their appropriate places. The iteration is the same as the loop in the previous section, but taking into account the placement on the game screen, variables a and b will represent the map variables created earlier, while variables x and y will represent the place where game objects will be placed in the game. If map[a, b] has a height minus 2, an exit object will be placed, otherwise a floor object will be placed. If map[x, y] is on the edge of the map, then the program will place an indestructible wall, otherwise a normal wall will be placed on the edge. If map[x, y] is worth 3, it is placed as a pawn, if it is worth 4, the program will place a pawn randomly from the existing list of pawns, if it is worth 5, the program will place an item from the existing list of items. After the looping process is complete the level is complete and ready to play.

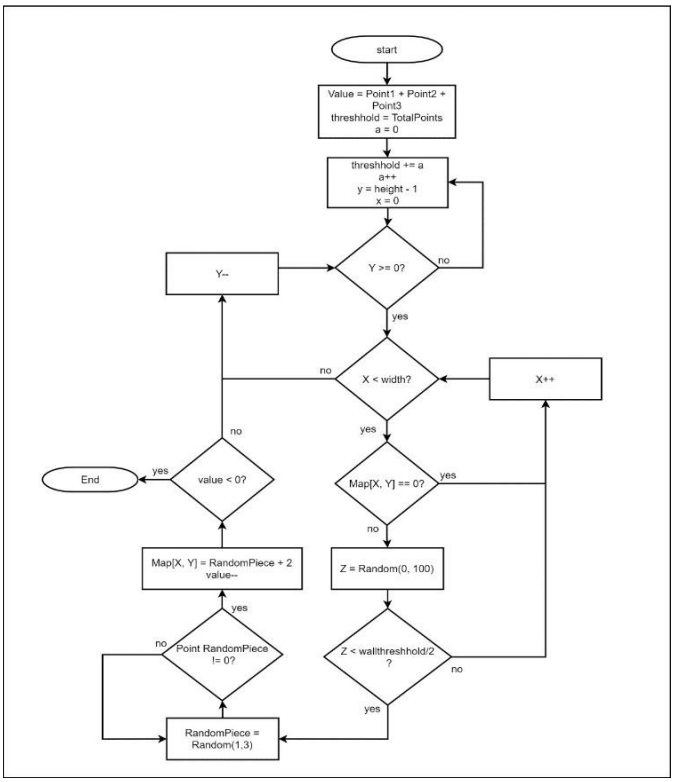

Fig. 6. FillGameSpace Chess Labyrinth Flowchart

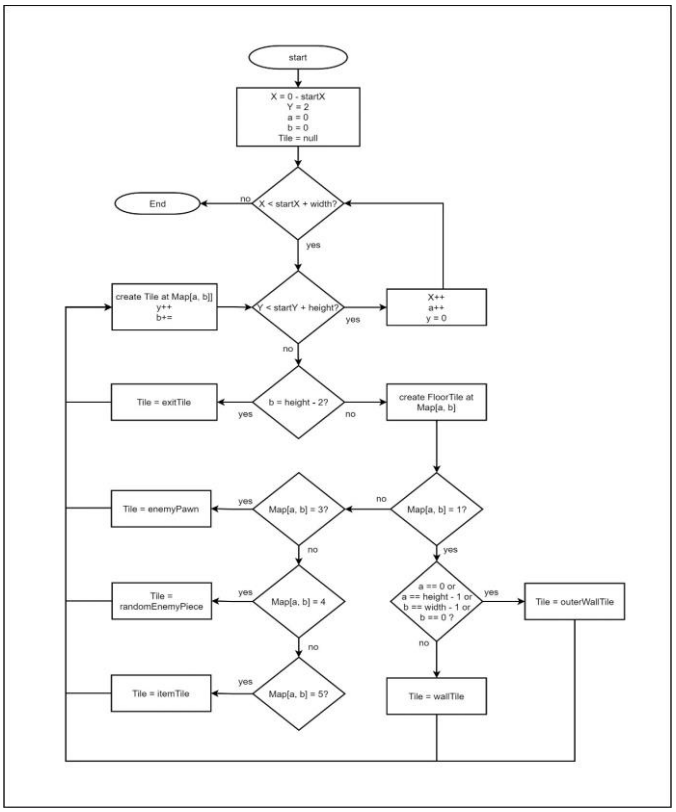

Fig. 7. Populate Map Chess Labyrinth Flowchart

Samuel Putra and Wirawan Istiono, "Implementation Simple Additive Weighting in Procedural Content Generation Strategy Game," *International Journal of Multidisciplinary Research and Publications (IJMRAP)*, Volume 4, Issue 12, pp. 9-18, 2022.

At the start of the game the level is set as 1 and a map of the level will be created based on what level it is at that time. If the level is more than 1 there will be a session to set the player's chess pieces, if it is still level 1 the player does not have money then this section will be skipped. If the level is completed it will be repeated again with increased level points. if the player loses or beats the last level then the game will be over and given the choice to start the game again or not, if selected not the app will close itself.

Players have various options to control the pieces they have, the options include moving the pieces, attacking the enemy or walls at a certain point, or activating the specific power of certain pieces. After the player's turn is finished the program checks whether the level is completed or not. If finished the player earns money and the level is completed, otherwise the enemy pieces controlled by the computer take their turn. The computer then executes a randomly determined action and the program checks whether the player loses or not. If the level is completed or the player loses the game that level is over.

The player chooses what action the piece will perform and then the program will perform the action according to the context at that time. If the player chooses to use a skill, which is a special ability that certain pieces have, the program will run the skill. If the player uses the knight's skill then the player will be able to move the knight piece and still be able to make a turn, if the rook's skill is activated then the player can place a wall on the level, if the queen's skill is activated then all enemies that are within the queen's reached will be attacked, and if the king's skill is activated then all the player's pieces close to the king will have their health points increase. If the player does not use a skill, the player can move the piece or attack the enemy if there is an enemy that can be attacked. If the player only wants to move the piece then the piece will move to the location chosen by the player. If the player chooses to attack the enemy, the piece will move to the closest position to the enemy being attacked. After the player performs an action then the player's turn ends and the enemy controlled by the computer will run.

First, the program will check whether any player's pieces can be attacked by enemy pieces. If there is, the enemy's piece will attack the player's piece and move to the closest position to the attacked piece. When there are no pieces to attack, the program will randomize the pieces and moves and then try to move the selected pieces to the selected moves. If the desired movement is impossible, for example if there is a wall blocking the movement, the random selection process will be repeated again. Every time the randomization process is carried out, the program adds one to a variable *a*, if the variable *a* reaches 30 then the program determines that no movement is possible and the enemy's turn will end without any action. Before the gameplay starts the player is given the opportunity to use the money earned from the gameplay session to help in the next level of the game. The player chooses the desired piece, if the piece is not owned, the player gets a new piece, and if there is already a piece, the level is increased. Players can start the game when it is finished.

Based on the level points you have, the difficulty and size of the map will change. The map is made with a 2-dimensional integer that will be filled with values that will be populated by the game object after the PCG process is completed by the program. If the player reaches level 30 then all these processes will be skipped and the player will play the last level that was created previously. Player strength is measured by 4 different values, which are translated into Point1, Point2, Point3, Point4. Point1 is the number of pieces owned by the player, Point2 is the level of each piece, Point3 is the health point of each piece, Point4 is the level point of the map to be created. Each piece also has a different value according to the level of use. All player pieces will be counted one by one and each point will be added according to the value of their respective importance. After each point is added up, all values are normalized and then calculated to produce a total score according to the importance of each point.

The total of the previous value is distributed into several categories, namely enemy pieces, number of walls, and other game items. Categories are selected randomly and after one category is selected the total value is reduced and then the next category is randomly selected again. This process is repeated continuously until the total value is exhausted. Each category has a different amount of importance, this process uses the SAW method but in reverse. After the total value is exhausted a variable called RandomFillWalls is filled with the result of Point4 to assist in the creation of the wall in the next part of the process.

The number of walls are made randomly with the help of the RandomFillWalls variable obtained from the previous process. If map[x, y] is at the edge then the value of map[x, y] will be 1. If map[x, y] is not at the end, the program will check randomly, if the random value obtained is less than RandomFillWalls then map[x, y] will be 1, otherwise it will be 0. This process will be repeated until all maps are filled with values. First the application will do a loop. Each iteration will check the neighbors of map[x, y]. if there is a neighbor with a value of 1 then the wallCount variable will increase by 1. If the total number of neighbors is more than 4 then map[x, y] will also be worth 1. If wallCount is less than 4 then map[x, y] will be worth 0.

The number of obstacles is obtained by adding up points 1-3 and the threshold using the total number of points obtained from the previous section, and then the variable a is used to speed up the looping process if random selection produces the same value over and over again. Then the same looping process with the placement of the walls in the previous section begins again. If map[x, y] is 0 then the program checks randomly, if the random value is more than the threshold map[x, y] is ignored and continues to the next iteration. If the random value is less than the threshold, an obstacle object will be assigned to that point. Then the obstacle is chosen randomly, if the previously defined obstacle is selected then map[x, y] will be filled with new value based on the corresponding obstacle value. If the selected hurdle is empty then random selection is started again until a non-blank value is obtained. When the loop is complete but not all obstacles are placed, the loop will start again from the beginning with

the threshold added by the value a to add the probability of the obstacle selected by a random program.

The map required by the game has been created, now all game objects are placed in their appropriate places. The iteration is the same as the loop in the previous section, but taking into account the placement on the game screen, the variables a and b will represent the map variables created earlier, while the x and y variables will represent the place where the game objects will be placed in the game. If map[a, b] has a height minus 2, an exit object will be placed, otherwise a floor object will be placed. If map[x, y] is on the edge of the map, then the program will place an indestructible wall, otherwise a normal wall will be placed on the edge. If map[x, y] is worth 3, it is placed a pawn, if it is worth 4, the program will place a piece randomly from the existing list of pieces, if it is worth 5, and the program will place an item from the existing list of items. After the looping process is complete the level is complete and ready to be played.

III. RESULT

The game detects what the player clicks by using Unity's RayCast feature. If the player selects a piece that the player has then the program will remember that piece and will initialize the available MoveTiles. The MoveTile is a visual representation of the move options that the player can make. AttackTiles serves to represent the choices of attacks that can be carried out by the player's pieces. AttackTile has the same method as MoveTile but only appears if there are enemies to attack. If a MoveTile is selected by the player, the program will send the location vector from the MoveTile and send it to the Move method in the pieces recorded by the program to initialize the movement process. After checking that there is no barrier between the player's piece and the goal, the piece will be moved by the SmoothMovement method. SmoothMovement serves to slow down movement to make it more visually appealing to players. If the player selects an AttackTile then the player will attack anything that is in the AttackTile position. If the enemy is attacked, the enemy's health points will be deducted and if it dies, the player gets money equal to the difficulty of the defeated enemy. If the object of the attack is a wall object, the wall will be destroyed.

First the generator adds level points and variables that will be used in subsequent reset methods. Afterwords the generator determines the width and height variables based on the level of the map. Then the generator creates a two-dimensional variable named map that uses the width and height dimensions. The generator then deletes all objects from the previous level that are child objects of the Map Generator object. If the player has reached level 30 the premade level will be created and if not, it will continue to the next method.

Before the map is formed, the program takes into account the level of difficulty of that level using the SAW method. The first program collects all the pieces owned by the player then checks each piece and calculates according to the type and condition of the piece. The importance of each criteria can be changed from the Unity editor to fasten the testing process. After the final score is obtained, the points are distributed to the level's obstacle points.

The obstacles the level has are randomly selected and reduce the final score. The seed and the useRandomSeed variables are used to perform the pseudo random seeding process. Pseudo random is a random type where all randomized values will be accessed. The result points determine the level to be created in the following process.

One of the points specified in the SAW Calculate section is used to create a RandomFillPercent variable that serves to fill in the wall points of the map. First, the application does seeding on pseudo random. The application then loops a number of widths and heights. In the loop, if x and y are on the edge of the map, then map[x, y] will have a value of 1. Then if not on the edge of the map, a pseudo random value will be checked and if the random result is less than the randomFillPercent variable then map[x, y] will be worth 1, and is 0 otherwise. After all the folders are filled, the application will run the smoothWalls process 5 times.

This method uses the cellular automata algorithm. First the application loops as much as the width and height variables. In the loop, the application checks the number of neighbors using the GetSurroundingWallCount method. The GetSurroundingWallCount method is used to see if the neighbor of map[x, y] in the map has a value of 1 or not, if the point around that point exceeds 4 then that point will also be worth 1. If the surrounding point is less than 4 then map[x, y] will be worth 0. After the map is tidied up the placement of other level obstacles can be done.

Then the program runs the RandomFillGameSpace method which functions to add obstacles to the map part that is still worth 0. First, the pseudo random initialization is the same as the previous method. After that, the maximum obstacles that the level has are calculated based on each point obtained from the previous SAW method. Then a testInt list variable is created from 1 to 3 with the same number value as the index in it. How to place obstacles using the same method as the method of placing walls above but RandomFillPercent is replaced by totalPoints obtained from the previous SAW method called wallThreshold. The variable a is used to add a wallThreshold if too many iterations occur. When pseudo random passes the wallThreshold, it will be randomized from 0 to the end of the intTest list. If a value is selected and the points corresponding to that point are still available then the resistance of that point is placed on map[x, y]. If the value of random is selected and the corresponding points are empty, the list will be reduced to ensure that it is not selected in the next random. After all occupied obstacle points have been placed into the map, the map is ready to be used by the program to initialize game objects.

The map variable is adjusted to the location vector in the game room, x and y represent the game room location vector, while a and b represent the index in the map variable. The program creates loops of x and y. If b is in a position before the top edge of the map, an exit object will be created which is always in the same place, otherwise one floor object will be randomly selected. The floor or exit point is at the bottom layer of other game objects, so after making a floor or exit, check the value of map[a,b]. Here are game objects that can be made

Samuel Putra and Wirawan Istiono, "Implementation Simple Additive Weighting in Procedural Content Generation Strategy Game," *International Journal of Multidisciplinary Research and Publications (IJMRAP)*, Volume 4, Issue 12, pp. 9-18, 2022.

Map[-, b] == (height-2) : created object exit
Map[a, b] == 0 : created a floor object
Map[a, b] == 1 : if at the edge of the map an indestructible wall is created, otherwise an ordinary wall is made
Map[a, b] == 3: created a pawn type enemy
Map[a, b] == 4: Created enemy of type bishop, knight, rook, or queen.
Map[a, b] == 5: a pickup object is created, which is an object that can give money to players

After all the map points are filled the level is ready to be played.

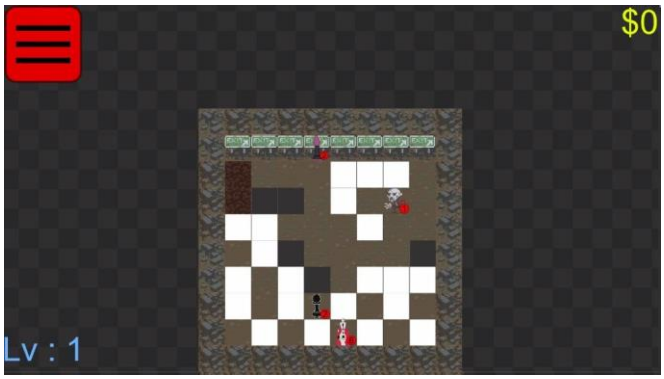

Fig. 8. Example Level 1 of Chess Labyrinth

Figure 8 is an example of a level 1 that can appear to players. The player starts each level by making the first turn. At the start of the game the player only has the king and pawns. Players can earn money for defeating enemy pieces, picking up game pickup objects, or completing levels. The money earned will be displayed on the top right of the screen.

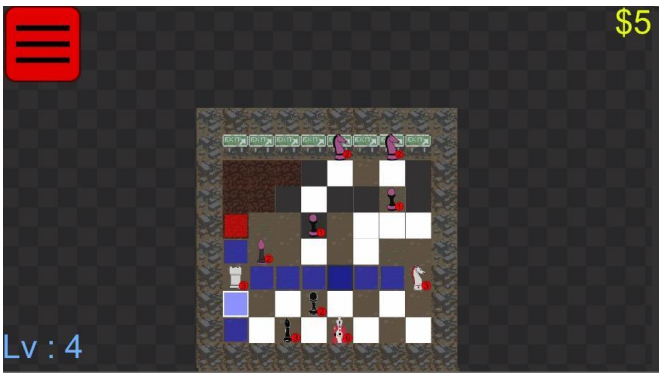

Fig. 9. Example of Chess Labyrinth Gameplay

Figure 9 is an example of the gameplay in the Chess Labyrinth game. Players can move a pawn by selecting the pawn they want to move, then the program will display the moves that the pawn can make. The program also displays if the player can attack a game object. Each type of pawn belonging to the player has a different movement and attack.

Figure 10 provides an example of the Army Management view. In this section the player can use the money collected during the game. Players can get new pieces by selecting a square that still doesn't have a pawn. Players can also increase the level of a pawn if they select a box that already has a pawn in it. Raising the level of a piece makes the status of the piece such as attack points, blood points, or others increase. After reaching a certain level a piece can get a special ability that is unique to that piece. After the player is ready to start the game, the player can click the start button which is under the money display at the top right of the screen.

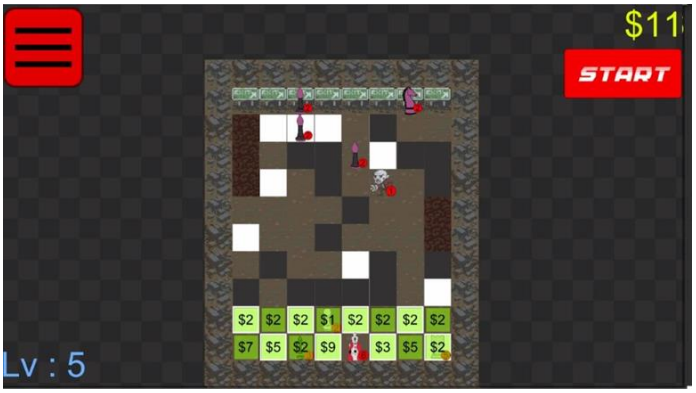

Fig. 10. Example Army Management session view of Chess Labyrinth

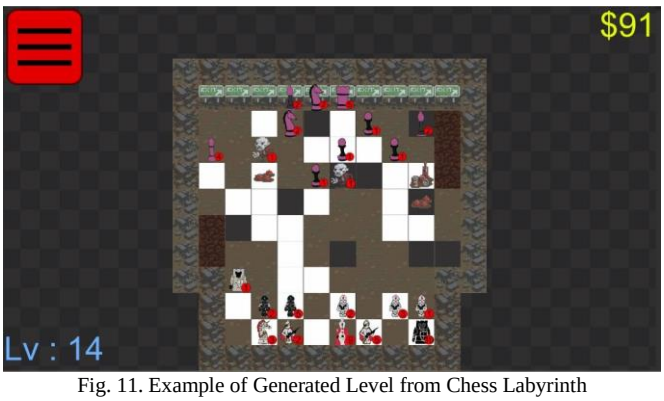

Fig. 11. Example of Generated Level from Chess Labyrinth

Figure 11 is an example of the levels created by PCG. Each time the player completes a level, PCG will create a new level with a difficulty level that matches the condition of the player's pieces. After a certain number of levels the created level will get bigger to increase the difficulty.

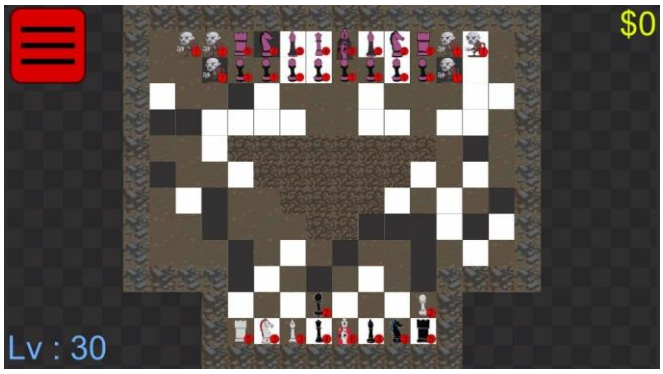

Fig. 12. Final Level Chess Labyrinth

Figure 12 shows the last level in the game. After reaching level 30 players will be given the last level as a final

Samuel Putra and Wirawan Istiono, "Implementation Simple Additive Weighting in Procedural Content Generation Strategy Game," *International Journal of Multidisciplinary Research and Publications (IJMRAP)*, Volume 4, Issue 12, pp. 9-18, 2022.

challenge. Players need to defeat enemy king pieces to win the game.

Prior to the implementation of the level map, it looked messy because it did not have the ability to detect the condition of the player at the level being played and could only take advantage of one variable, namely the value of the level being played, because of these limitations the map creation was made with filling that did not match the player's condition. After implementing PCG, you can create a level map according to the player's condition because it is affected by various additional variables such as the number of player pieces, the types of player pieces or the number of health points of each piece. But to determine changes in user experience, a more proven evaluation method is needed.

Fig. 13. PCG Comparison

Figure 13 shows the PCG generated before and after the implementation of SAW with various sizes and levels of difficulty. The image on the left is before the SAW implementation and the right is after. By looking at this example, it can be seen that the changes that occurred after the implementation of SAW, before the implementation of the level map looked messy because it did not have the ability to detect the condition of the player at the level being played and could only take advantage of one variable, namely the level of the level being played, because of these limitations, map making made with filling that is not suitable for the condition of the player. After implementing PCG, you can create a level map according to the player's condition because it is affected by various additional variables such as the number of player pieces, the types of player pieces or the number of blood points of each piece. But to determine changes to the user experience, a more proven evaluation method is needed. When making PCG it is important to do an evaluation. Evaluation is important to see the success between assets and players. The research also states the use of the Game Experience Questionnaire in the form of a questionnaire to evaluate player experience [12].

Thirty respondents were asked to play the game before and after the implementation of SAW was applied to PCG after being explained how to play the game and providing feedback in the form of answering a questionnaire. The questionnaire used is the Game Experience Questionnaire which has seven different categories, this study will use four of the following categories. The four points are Competence, Challenge, Negative Effect, and Positive Effect [13], [14]. Competence variable is used to assess the skill level of players. Challenge variable to assess whether players feel challenged. Positive effect to find out the positive elements experienced by players during the game. Negative effect to find out the negative elements that may be experienced by players. The questions contained in the questionnaire are in the form of a collection of statements where the player fills in whether the player agrees with the statement or not. The question list of questionnaire can be seen in Table I.

TABLE I. LIST OF QUESTION.

Competence category questions:
• I feel i can play the game without any problems
• I feel good/competent at the game
• I have no problem finding the right solution during the game
• I solve problems in the game quickly
Challenge category questions:
• I find the game difficult
• I often feel trapped in difficult situations during the game
• I feel challenged
Positive Effect category questions:
• I find the game fun
• I feel satisfied with the game
• I enjoy my time when i play
• I am interested in playing the game more than once
Negative Effect category questions:
• I feel bored during the game
• I find playing the game tiring
• I think of other things while playing the game
• I'm not interested in playing the game

In the questionnaire, there is five section that can be choosen by the respondent to measure the level of user satisfaction, such as strongly disagree (SD), disagree (D),

neutral (N), agree (A), or strongly agree (SA). And the evaluation result can be seen in Table II.

TABLE II. COMPETENCE EVALUATION

Before Implementation						
No	SD	D	N	A	SA	Average
1	8	12	1	7	2	42%
2	5	14	2	7	2	51.33%
3	14	7	1	6	2	43.33%
4	7	12	2	6	3	50.66%
Final Result						46.83%
After Implementation						
No	SD	D	N	A	SA	Average
1	2	7	6	7	8	68%
2	2	7	6	7	8	68%
3	2	6	10	5	7	66%
4	3	7	7	6	7	64.66%
Final Result						66.66%

Table II shown the result of the Competence section of the questionnaire. According to the results of the questionnaire before the implementation of SAW the player's competency value was 46.83% and after the implementation of SAW the same player got a score of 66.66%. After the implementation of SAW, the players felt that their competence increased by 19.83%.

TABLE III. CHALLENGE EVALUATION

Before Implementation						
No	SD	D	N	A	SA	Average
5	2	3	2	13	10	77.33%
6	1	6	5	6	14	82%
7	1	3	4	14	8	76.66%
Final Result						78.66%
After Implementation						
No	SD	D	N	A	SA	Average
5	4	3	3	14	6	70%
6	10	12	0	7	1	44.66%
7	5	12	2	8	3	55.33%
Final Result						56.66%

Table III shown the result of the Challenge section of the questionnaire. According to the results of the questionnaire before the implementation of SAW the difficulty value of the game according to the players was 78.66% and after the implementation of SAW the players rated the difficulty of the game by 56.66%. According to players after the implementation of SAW the game difficulty was reduced by 22%.

TABLE IV. POSITIVE EFFECT EVALUATION

Before Implementation						
No	SD	D	N	A	SA	Average
8	6	10	4	9	1	52.66%
9	4	17	2	6	1	48.66%
10	10	8	3	7	2	48.66%
11	6	15	3	6	0	46%
Final Result						49%
After Implementation						
No	SD	D	N	A	SA	Average
8	1	1	3	21	4	77.33%
9	2	1	5	17	5	74.66%
10	2	6	9	5	8	67.33%
11	1	2	10	14	3	70.66%
Final Result						72.5%

Table IV shown the result of the Positive Effect section of the questionnaire. According to the results of the questionnaire before the implementation of SAW, the positive elements obtained by players during the game were worth 49% and after the implementation of SAW the positive elements obtained by players were worth 72.5%. According to players after the implementation of SAW the positive elements increased by 23.5%.

TABLE V. NEGATIVE EFFECT EVALUATION

Before Implementation						
No	SD	D	N	A	SA	Average
12	2	6	2	11	9	72.66%
13	1	3	3	17	6	76%
14	0	8	6	9	7	70%
15	1	8	5	8	8	69.33%
Final Result						72%
After Implementation						
No	SD	D	N	A	SA	Average
12	6	16	0	7	1	47.33%
13	7	18	0	4	1	40%
14	8	13	4	4	1	42%
15	13	10	3	3	1	39.33%
Final Result						42.17%

Table V shown the result of the Negative Effect section of the questionnaire. According to the results of the questionnaire before the implementation of SAW, the negative elements obtained by players during the game were worth 72% and after the implementation of SAW, the negative elements obtained by players were worth 42.17%. According to players after the implementation of SAW the negative elements were reduced by 29.83%.

IV. FUTURE WORK

In making games using PCG, it is recommended to prepare a large variety of game objects in order to avoid game content looking too similar to each other, even though game objects have the same function, differences in visual appearance can make the game more visually attractive. It is recommended not to use the Cellular Automata algorithm to assist in the creation of small-scale vector maps, because the effectiveness of the algorithm decreases with the size of the map. In making strategy games, it is advisable to emphasize the design of comprehensive tutorials to teach users the systems to be used and to create artificial intelligence that can adjust to the level of difficulty of the level being played so that players can learn the game system when the level is easier to complete.

ACKNOWLEDGMENT

Thank you to the Universitas Multimedia Nusantara, Indonesia which has become a place for researchers to develop this journal research. Hopefully, this research can make a major contribution to the advancement of technology in Indonesia."

REFERENCES

[1] A. N. Foster, "The process of learning in a simulation strategy game: Disciplinary knowledge construction," *Journal of Educational Computing Research*, vol. 45, no. 1, pp. 1–27, 2011.
[2] S. Dor, "Strategy in games or strategy games: Dictionary and

encyclopaedic definitions for game studies," *Game Studies*, vol. 18, no. 1, 2018.
[3] A. Y. C. Leong, M. H. Yong, and M. H. Lin, "The effect of strategy game types on inhibition," *Psychological Research*, no. 0123456789, 2022.
[4] D. Hooshyar, M. Yousefi, and H. Lim, "A Procedural Content Generation-Based Framework for Educational Games: Toward a Tailored Data-Driven Game for Developing Early English Reading Skills," *Journal of Educational Computing Research*, vol. 56, no. 2, pp. 293–310, 2018.
[5] J. Aycock, *Procedural Content Generation in Games*. Switzerland: Computational Synthesis and Creative Systems, 2016.
[6] J. Togelius *et al.*, "Procedural Content Generation : Goals, Challenges and Actionable Steps," *Artificial and Computational Intelligence in Games*, vol. 6, no. July, pp. 61–75, 2013.
[7] W. Istiono, "Does the Education Games with adding some Entertainment Game Elements will attract the children?," *International Journal of Advanced Trends in Computer Science and Engineering*, vol. 10, no. 4, pp. 2721–2726, 2021.
[8] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup, "Procedural content generation for games: A survey," *ACM Transactions on Multimedia Computing, Communications and Applications*, vol. 9, no. 1, 2013.
[9] A. W. Istiono and A. Waworuntu, "What element that influence preschool and elementary school children to enjoy playing education games ?," *International Journal of Advanced Studies*, vol. 9, no. 12, pp. 9–13, 2021.
[10] D. Wira Trise Putra and A. Agustian Punggara, "Comparison Analysis of Simple Additive Weighting (SAW) and Weigthed Product (WP) in Decision Support Systems," *MATEC Web of Conferences*, vol. 215, pp. 1–5, 2018.
[11] A. Setyawan, F. Y. Arini, and I. Akhlis, "Comparative Analysis of Simple Additive Weighting Method and Weighted Product Method to New Employee Recruitment Decision Support System (DSS) at PT. Warta Media Nusantara," *Scientific Journal of Informatics*, vol. 4, no. 1, pp. 34–42, 2017.
[12] D. M. De Carli, F. Bevilacqua, C. T. Pozzer, and M. C. D'Ornellas, "A survey of procedural content generation techniques suitable to game development," *Brazilian Symposium on Games and Digital Entertainment, SBGAMES*, no. September 2018, pp. 26–35, 2011.
[13] M. H. Phan, J. R. Keebler, and B. S. Chaparro, "The Development and Validation of the Game User Experience Satisfaction Scale (GUESS)," *Human Factors*, vol. 58, no. 8, pp. 1217–1247, 2016.
[14] K. L. Norman, "GEQ (Game Engagement/experience questionnaire): A review of two papers," *Interacting with Computers*, vol. 25, no. 4, pp. 278–283, 2013.