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

Implementation Drunkard's Walk Algorithm to Generate Random Level in Roguelike Games

Andy Koesnaedi[1], Wirawan Istiono[2]
[1,2]Informatic, Universitas Multimedia Nusantara, Tangerang, Banten, Indonesia
Email address: andy.koesnadi@gmail.com, wirawan.istiono@umn.ac.id

Abstract— *Video games are no stranger to us listening to. Video games are one of the rapidly growing entertainment industries. Video Games content request are always evolving but making video games is not easy and also resource-consuming. One way to prevent this problem is by using Procedural Content Generation or known as PCG. By using PCG, the replaybility of level in a video game increase. The algorithm used in the application of this PCG is Drunkard's Walk. Level is generated randomly by using this algorithm. This study aims to design and build a video game with roguelike genre using the Drunkard's Walk algorithm, which was created with using Unity game engine. After finishing building the game, then player satisfaction level is measured by using GUESS or Guest User Satisfaction Scale. By using GUESS as a measuring tool, 35 respondents successfully obtained. From 35 respondents, the value obtained using GUESS as a benchmark is 84.58% which is very good.*

Keywords— *Drunkard's Walk, Guest User Satisfaction Scale, Procedural Content Generation, Video game.*

I. INTRODUCTION

The content from video games is an important factor why video games as mentioned continue to be played every day. However, the demand for new content continues to grow and to produce new content is already expensive and unmeasured. Lots of video games are fun to play for the first time, but if the video game has a fixed problem solving and, in some skill, levels only have one same playthrough, just one finished playthrough will take away the attraction of playing that video games [1], [2]. One way to prevent the loss of attraction is by creating content that has replaybility characteristic that aim for each playthrough will always be different from the previous one, and to create this method will be use Procedural Content Generation method.

Roguelike game based on 2 games which are dungeon questing games, namely Rogue in 1980 and Moria in 1983. Procedural Content Generation has been used in these two games, which have become the basic concept of roguelikes to this day [3]. Roguelikes have levels that have several rooms, of varying sizes and shapes, connected by corridors until each room or corridor has been positioned. This room contains by variety of monsters, treasures, weapons and traps [4].

Procedural Content Generation is one of the factors in making roguelike games. By using Procedural Content Generation [5], [6], then Algorithms such as level building and monster laying will be carried out automatic. Drunkard's Walk is one of the simplest generators available that will generate a level like a cave [7]. This algorithm works by walking randomly to make a pattern like a drunk person.

Cellular Automata is one of the generators that a cell will determine the number of cells that will live from the surrounding cells [8]. If cell that live too much or too little, the cell will then die [9]. The disadvantage of the Cellular Automata algorithm is that if it is in the process, generation there are many walls [10], then the playable space for players will be small. And, if Cellular Automata does multiple generations [11], the number of rooms and corridors will also decrease, but the size of the corridors will be consistent and the size of the room will get bigger [12]. Meanwhile, in the Drunkard's walk algorithm generation, the more generations, and the more generated playable space of the room. Moreover, more generations will increase the size of the corridor, but the size of the room will remain the same [13]. The reason this game uses the Drunkard's Walk algorithm is because the Drunkard's Walk algorithm will produce a map that will merge and have a start and finish, while the Cellular Automata algorithm will generate a map that is difficult to generate a map that has a room that has relationship [14].

Based on the introduction above, the design and development of a roguelike themed game will use the Drunkard's Walk algorithm, which will result in Procedural Content Generation. This algorithm will make connections between rooms that will be generated by the algorithm. After completing the game, the author will distribute the game to players who like roguelike genre games. After spreading in the game that will be designed and created, these players will be asked to fill out a questionnaire using Game User Experience Satisfaction (GUESS) to measure player satisfaction.

II. LITERATURE STUDY

A. *Roguelike*

The beginning of Roguelike originated from a game called Rogue in 1980. Rogue is a game with a turn-based dungeon crawler genre, where the player must pass through several dungeons, retrieve items that are exist, and defeat the monsters. Rogue is designed using simple ASCII graphics to generate enemies and rooms [4].

Fig. 1. Rogue game screen

Figure 1 shown a Rogue game screen made in 1980. This game using Procedural Content Generation which was one of the initial concept's roguelike games. Rogue is made with a design that uses ASCII as a visual [4].

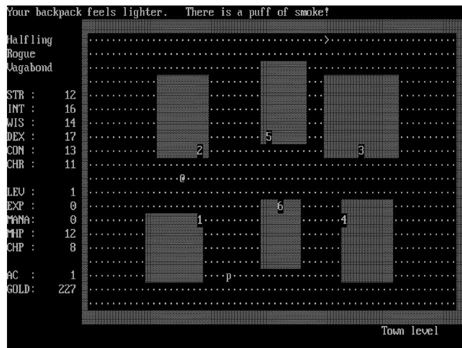

Fig. 2. Moria game screen

Figure 2 shown a Moria game screen created in 1983. This game is also using Procedural Content Generation and also become one of the concepts early roguelike games [4], [15].

B. *Procedural Content Generation*

Procedural Content Generation is a media generator that works automatically. Media that produce textures, sound effects, maps, levels, characters, quests, and also the mechanics of the game [16]. Game content that can be generated can be divided into 6 parts, namely:

- Game Bits, that is the basic unit in a game content. game bits needed in the game, such as the texture that distinguishes the main character and enemies, or like background music, but game bits don't interact directly with players. Examples of game bits are textures, sounds, effects textures (fire, water, wind), and others.
- Game Space, which is an environment in which the game takes place, including maps and terrain.
- Game Systems, which simulate more complex environments, such as ecosystems, road networks, urban environments.
- Game Scenarios, set the previous level so that it becomes coherent or be a sequence of events. Game Scenarios includes puzzles, stories and levels.
- Game Design, namely a rule and mechanism contained in the game.
- Derived Content, can be created which is useful as a companion in games. Examples of derived content are leaderboard and flavor text which will be useful to help players.

From the game content described above, the Procedural Content Generation method is classified into 3, namely [14], [17]:

1. Traditional Methods are pseudo-random number generators which were used by the very first commercial video games and have been generally used for dungeon and labyrinth generation. The first advantage of using this method is that it is simple and fast. An example of using traditional methods is to generate fractals and noise that can create vegetation.
2. Search Based Methods perform content generation and then evaluate it. Usually, this method is divided into components, namely space representation, reachability evaluation, and search algorithm.
3. Machine Learning is used to classify or predict existing problems.

C. *Drunkard's Walk*

Drunkard's Walk or what can also be called random walk is the most basic rhythm algorithm used to generate levels like caves. Drunkard's walk algorithm works by choosing a random point and move randomly [18]. This path is repeated until it reaches the desired level.

$$S_n = \sum_{i=1}^{N} X_1 + \cdots + X_n$$

Formula above is a random walk formula. Sn is the number of steps the random moving path of Xi's steps, then Sn will form a path from number of steps Xi [17].

$$S_{t=1} = S_t + W_t$$

Formula above is still a random walk formula. S_t is existence the current location is based on the value of t, and W_t is a step or random variable with the distribution value.

III. METHODOLOGY

A. *Research Methodology*

The research methodology that will be used in the development of this game, first step is study literature, where in this stage, research is done before starting the design of this game made. The research sought is in the form of designing a roguelike game using Procedural Content Generation with the Drunkard's Walk algorithm and GUESS-18. And the second step is Designing Game, where in this stage the author will design a Game Design Document, which aims to describe the details of the game to be designed. The third step is creating game, where In making this game, it will be made with the Unity game engine 2020.3.26f1. and Visual Studio Code 2019 which acts as an IDE using the C# programming language. For the operating system, Windows 10 will be used. The forth step is testing application, where in testing will be done after game development is complete. Testing will be carried out with a minimum of 30 players, and player satisfaction will be measured using GUESS-18. The next step is evaluation, after the development and testing stages are completed, the available data will be collected, and a conclusion will be drawn on the level of player satisfaction of the games that have been designed. And the last step is documentation, where the form of documentation that will be carried out is in the form of screenshots or important notes.

B. *Game Design*

The game that will be built is named Monster Cave. The design of this game will be shown using flowchart. Figure 3 is a flowchart of the main menu. When the player starts game, players will see the splash screen first before heading to main menu. When in the main menu, players can choose the new option games, credits, and exits. If the player selects new game, the game will display the story first, and after the story

is displayed, the player can choose to play right away or see how to play first. If the player selects credits, the player can see the credits page will display the assets used in this game. And if the player chooses exit, then the game will close itself.

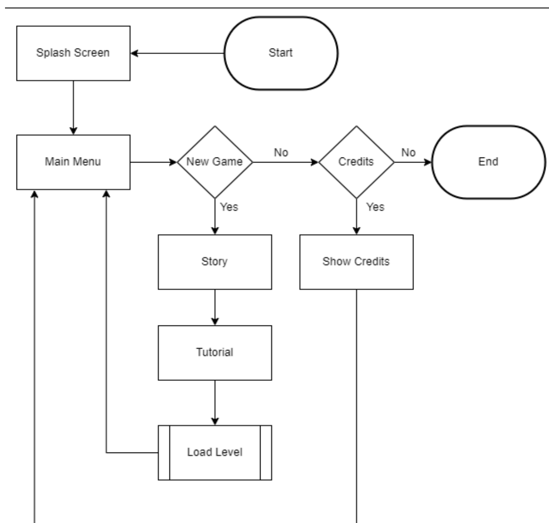

Fig. 3. Flowchart Main Menu

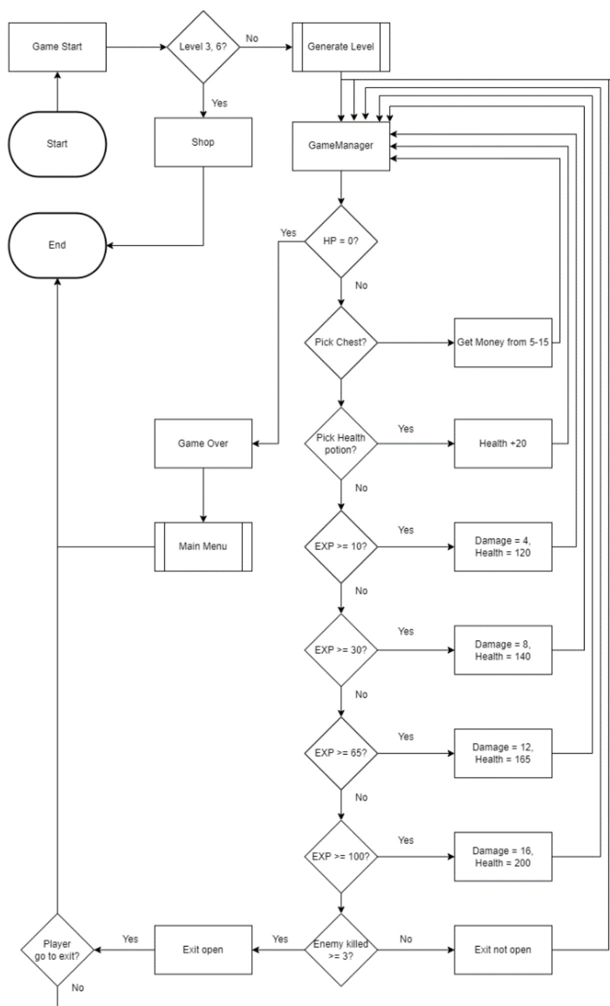

Fig. 4. Flowchart Gameplay

Figure 4 shown a gameplay flowchart. Before the game starts, the game will check what level to run. If the level will run is level 3 or level 6, then the shop will appear. If the executed level is neither level 3 nor level 6, then the level will be generated.

After the level has been generated, the game manager will run and look for players. Game manager will load information in the form of players HP, player experience, and money earned when playing. After the information has been loaded, the camera will look for the whereabouts of the player and will continue to follow him. The HUD will also be displayed, which will display the HP, experience, and money earned.

In Figure 4 also shown, the game will check the player's HP, if the player's HP reaches 0 then the game over display will come out, and players will be redirected to the main menu. Players can replenish HP by getting health potion that is dropped if you kill an enemy or player can buy it in a shop, which will fill 20 HP. Players will gain experience if enemies are killed or player can buy experience in the shop.

If the player's experience has not reached 10, the player's damage begins of 1 and HP of 100. If the player's experience reaches 10, then the damage will be increased to 4 and the HP will be 120. If the experience the player reaches 30, then the damage will be increased to 8 and HP becomes 140. If the player's experience reaches 65, the damage will be increased to 12 and the HP will be 165. If the player's experience reaches 100, the damage will be increased to 16 and the HP will be 200.

To go to the next level, the player must kill at least 3 enemies first to go to the next level. If players directly walk to the exit, players cannot go to the level next and the game will show that the player must kill at least 3 enemies first. If the player has killed 3 enemies and the player walks to the exit, then the next level will be generated.

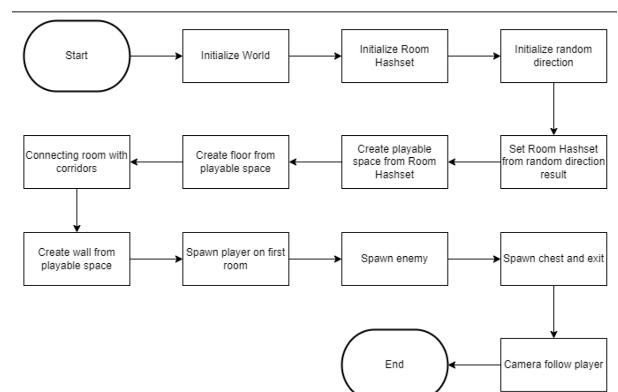

Fig. 5. Flowchart Level Generator

Figure 5 shown a flowchart level generation, which explains how the drunkard's walk algorithm works to create a map for this game at random. First a room hash set will be created and will define the initial position for the walker. After that the walker will run automatically random which will then be stored in the hash set.

After completing the measurements, the room hash set that has been collected by this walker will then be used as the floor as an area play player. After the floor has been made, the

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

walker will walk make another room. After finishing making some rooms, then the rooms that are made will be connected using the corridor.

After the room and corridor are finished, the walls will made around the room and corridor. If in the room there is the hole in the floor, then the hole will be filled with walls. After the wall is completed, the player will be shown to the first room created. After that the enemy will appear in a room other than the player's room. After the enemy has appeared then the chest and an exit will appear. After that the camera will look for where the player is and will follow the player.

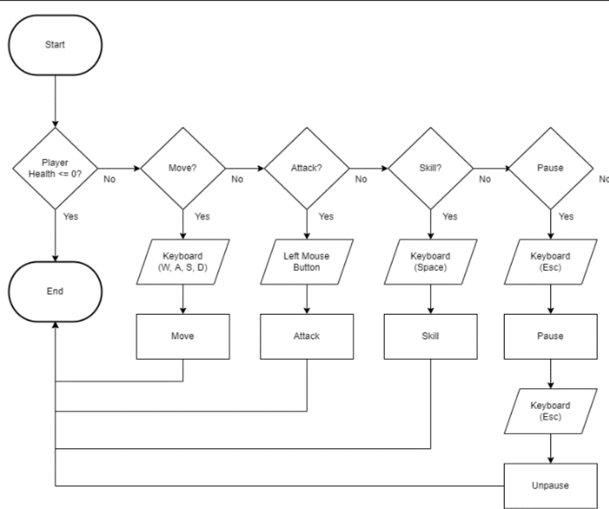

Fig. 6. Flowchart Player

Figure 6 shown flowchart player, where the game will keep checking whether the player has HP or not. The game will continue if the player still has HP, and the game will be over if the HP the player has reached 0. The player has several inputs, for moving player can enter keyboard input in the form of w, a, s, and d. To perform an attack the player can enter the input left mouse on the mouse. To activate the skill that is to make the player cannot receive attacks, players can enter keyboard input in the form of space. To display the pause menu, players can enter the keyboard input in the form of escape. The player can exit the game if the player selects the quit button, and the player can continue the game if the player selects the resume button.

Figure 7 shown flowchart of the enemy where the game will give HP and damage to the enemy and will continue to check whether the enemy's HP has reached 0 or not. If the enemy's HP reaches 0, then the enemy will be eliminated, and if the enemy still has HP, then the enemy will stay in the game. The enemy will continue to look for players around radius, and if the player is already in the enemy's radius, then the enemy will chase the players. If the enemy is already within the attack radius, then the enemy will attack the player. If the player manages to escape from the enemy, the enemy will return to their original position.

To implementation drunkard's walk algorithm in this game, first step is preparing some global variable that can be catch by other script, to set variable global in this case, will be use playerPref method that can be seen in sample code in Figure 8.

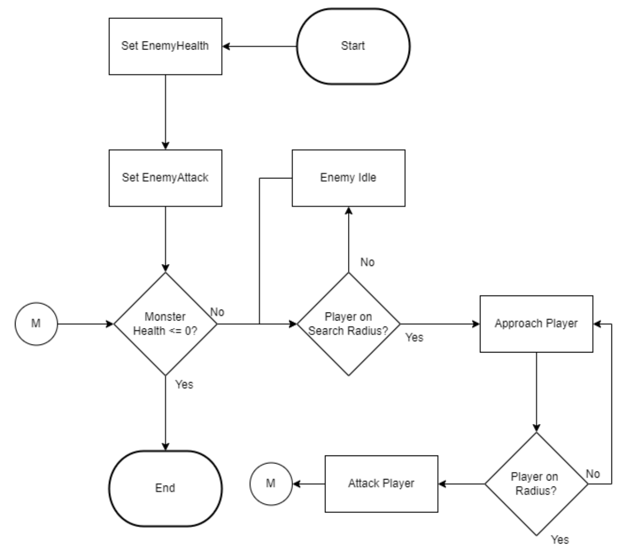

Fig. 7. Flowchart Enemy

```
3 references
public void SaveState()
{
    string sving = "";

    sving += "0" + "|"; //Empty
    sving += money.ToString() + "|"; //PlayerMoney
    sving += experience.ToString() + "|"; //PlayerEXP
    sving += hitPoint.ToString() + "|"; // PlayerHP

    PlayerPrefs.SetString("SaveState", sving);
}
```

Fig. 8. Playerprefs Implementation

Figure 8 shown a code snippet that uses Playerprefs to store player data when changing levels. Player data that will be saved is the money collected, player experience that have been collected, and the player's HP.

```
2 references
public void LoadState(Scene sving, LoadSceneMode mode)
{
    if (!PlayerPrefs.HasKey("SaveState")){
        return;
    }
    SceneManager.sceneLoaded -= LoadState;
    Debug.Log("Loading save file");

    string[] data = PlayerPrefs.GetString("SaveState").Split('|');

    money = int.Parse(data[1]);
    experience = int.Parse(data[2]);
    hitPoint = int.Parse(data[3]);
}
```

Fig. 9. Playerprefs Implementation

Figure 9 shown a code snippet that uses Playerprefs to load player data when starting a new level. Started by reading a key named SaveState, which is then a collection of arrays string will be parsed and loaded in each value.

Figure 10 shown a code snippet that shows the function of levels. It can be seen if experience is below 10, then the function is level 1 will be called. If experience is below 30,

then level 2 functions will be called, and so on. If experience has reached 100, then level players reach the maximum and can no longer get experience again.

```
1 reference
void level()
{
    if (gameManager.experience < 10) //Level 1
    {
        xptext.text = "XP : " + gameManager.experience.ToString() + " / 10";
        maxHitPoint = 100;
        playerWeapon.weaponDamage = 1;
    }
    else if (gameManager.experience < 30) //Level 2
    {
        xptext.text = "XP : " + gameManager.experience.ToString() + " / 30";
        maxHitPoint = 120;
        playerWeapon.weaponDamage = 4;
    }
    else if (gameManager.experience < 65) //Level 3
    {
        xptext.text = "XP : " + gameManager.experience.ToString() + " / 65";
        maxHitPoint = 140;
        playerWeapon.weaponDamage = 8;
    }
    else if (gameManager.experience < 100) //Level 4
    {
        xptext.text = "XP : " + gameManager.experience.ToString() + " / 100";
        maxHitPoint = 165;
        playerWeapon.weaponDamage = 12;
    }
    else if (gameManager.experience >= 100) //Max Level
    {
        xptext.text = "XP : 100 / 100";
        maxHitPoint = 200;
        playerWeapon.weaponDamage = 16;
    }
}
```

Fig. 10. Player Level Implementation

```
1 reference
public static HashSet<Vector2Int> DrunkardWalk(Vector2Int startPosition, int drunkWalkLength){
    HashSet<Vector2Int> path = new HashSet<Vector2Int>();

    //Starting Position
    path.Add(startPosition);
    var previousPosition = startPosition;

    for (int i = 0; i < drunkWalkLength; i++){
        //Implementing Random Walk
        var newPosition = previousPosition + Direction2D.GetRandomDirection();
        path.Add(newPosition);
        previousPosition = newPosition;
    }
    return path;
}
```

Fig. 11. Drunkard's Walk Algorithm

```
public static class Direction2D{
    // Method untuk random direction
    2 references
    public static Vector2Int GetRandomDirection()
    {
        return randomDirectionsList[Random.Range(0, randomDirectionsList.Count)];
    }

    public static List<Vector2Int> randomDirectionsList = new List<Vector2Int>{
        new Vector2Int(0, 1), //UP
        new Vector2Int(1, 0), //Right
        new Vector2Int(0, -1), //Down
        new Vector2Int(-1, 0) //Left
    };
```

Fig. 12. Drunkard's Walk Algorithm Snippet

Figure 11 shown a Drunkard's Walk code snippet. This code will save the initial position. In the for function, the newPosition variable will run randomly from its initial position, which is called on the Direction2D function contained in the code snippet that can be seen in Figure 12. After newPosition has run, the position will be stored in the path and will be save new position as starting position

```
3 references
public void PaintFloorTiles(IEnumerable<Vector2Int> floorPositions){
    PaintTiles(floorPositions, floorTilemap, floorTile);
}

1 reference
private void PaintTiles(IEnumerable<Vector2Int> positions, Tilemap tilemap, TileBase tile){
    foreach (var position in positions)
    {
        PaintSingleTile(tilemap, tile, position);
    }
}
3 references
private void PaintSingleTile(Tilemap tilemap, TileBase tile, Vector2Int position)
{
    var tilePosition = tilemap.WorldToCell((Vector3Int)position);
    tilemap.SetTile(tilePosition, tile);
}
```

Fig. 13. Snippet of Code Making Map

Figure 13 is a code snippet for creating a map. Floor positions will retrieve position information from the Drunkard's Walk function implemented in Figure 11, and will be combined to the floorPositions hash set. After that, the information will be given to the Paint FloorTiles function which is used to create floors and walls using tilemap.

```
public class MapGenerator : DungeonGenerator
{
    [SerializeField]
    protected RandomWalkSO randomWalkParamaters;

    4 references
    protected override void ProceduralGeneration(){
        //Create floor position
        HashSet<Vector2Int> floorPositions = RunRandomWalk(randomWalkParamaters, startPosition);
        tilemapGenerator.Clear();

        tilemapGenerator.PaintFloorTiles(floorPositions); //Make floor for dungeon
        WallGenerator.CreateWall(floorPositions, tilemapGenerator); // Make wall for dungeon
    }

    4 references
    protected HashSet<Vector2Int> RunRandomWalk(RandomWalkSO parameters, Vector2Int position)
    {
        var currentPosition = position;
        HashSet<Vector2Int> floorPositions = new HashSet<Vector2Int>();

        //Making floor position
        for (int i = 0; i < parameters.iterations; i++){
            var path =  ProceduralAlgorithm.DrunkardWalk(currentPosition, parameters.walkLength);
            floorPositions.UnionWith(path);
            if(parameters.randomIteration)
            {
                currentPosition = floorPositions.ElementAt(Random.Range(0, floorPositions.Count));
            }
        }
        return floorPositions;
    }
}
```

Fig. 14. Snippet of Code Making Floor

Figure 14 is a code snippet for creating a floor. Information floorpositions will be taken from the function in figure 13, and then the tilePosition variable will retrieve the position information and will be entered tilemap to the position

```
2 references
private static HashSet<Vector2Int> FindWallsInDirections(HashSet<Vector2Int> floorPositions, List<Vector2Int> directionList)
{
    HashSet<Vector2Int> wallPositions = new HashSet<Vector2Int>();
    foreach (var position in floorPositions)
    {
        foreach (var direction in directionList)
        {
            //Cheking floor position
            var neighbourPosition = position + direction;

            //if floorPositions doesnt have neighbourPosition, it is a wall
            if (floorPositions.Contains(neighbourPosition) == false)
            {
                wallPositions.Add(neighbourPosition);
            }
        }
    }
    return wallPositions;
}
```

Fig. 15. Snipper of Code Making Wall

Figure 15 shown a code snippet for creating walls. Newly hash set created to prevent duplicates. To search the wall, every position will be examined by the board of directors. If the position being checked does not exist directors who have a floor that is variable neighborPosition, then the position is the wall and will be entered into a hash set named wall Positions. These WallPositions will become walls.

IV. RESULT

From the methodology and code that has been applied to the Unity engine application with C# code, the results can be seen from Figure 17 to Figure 24.

Figure 16 shown the main menu that will appear after the splash screen. In the main menu, players can start a new game, view credits, or exit the game. Form main menu, the game will show the story of this game when players choose the New Game button, and also after the story shown, a tutorial how to play this game will be shown. To go back to the main menu,

when player in gameplay screen or credits screen, players can press the back button.

Fig. 16. Main Menu Screen

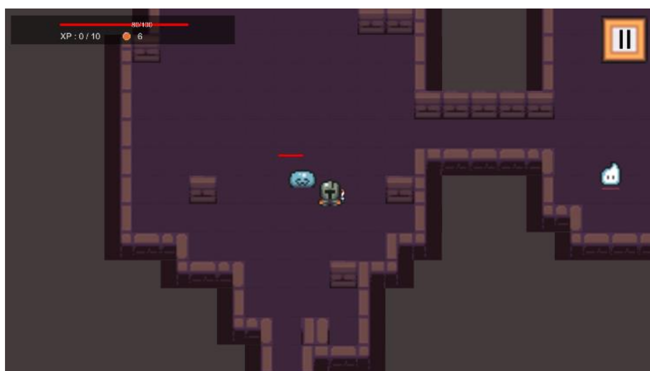
Fig. 17. Gameplay

Figure 17 shown sample of the gameplay that implements Drunkard's walk algorithm. Where the players can see HP, experience, money, and pause button. Character's HP will decrease if hit by an attack from enemy and will increase if character take a health potion.

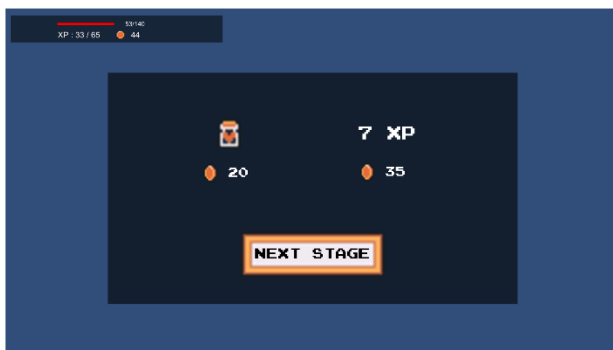

Fig. 18. Shop screen

To make this game more interesting, beside adding Drunkard's walk algorithm, this game also adding shoping menu that can be seen in Figure 18. The shop provides health potions and experience items to buy by a player.

Figure 19 shown the credit screen, where the players can see credits from this games. The credit contains the creator of this game and the creator of the assets used in the making of this game.

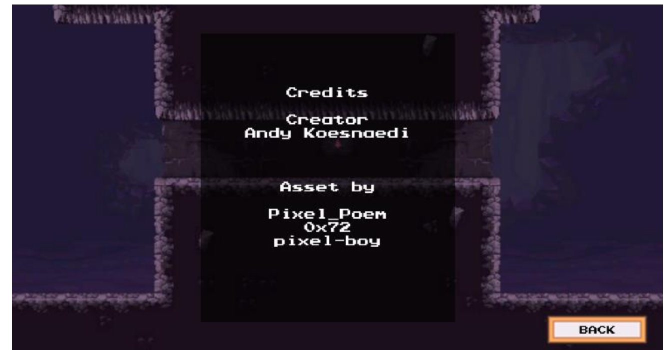

Fig. 19. Credits

After making a game, user acceptance testing is carried out by GUESS method, the Questionnaire survey are given to players who have played the game with Drunkard's walk algorithm, for measuring the level of player satisfaction. The Questionnaire have 18 questions with seven grades of assessment categories. Categories rated by GUESS are Narratives, Play Engagement, Enjoyment, Creative Freedom, Audio Aesthetic, Personal Gratification, Social Activity, and Visual Aesthetic, but because this game is single-player and has no sound, so the question is only 13 questions are available. The questionnaire was filled out by 35 respondents from various ages, from children, teenagers to adults, where 88.6% of respondents who filled out the questionnaire were aged 20 to 30 years old, the respondent's age can be seen in Figure 20.

Umur

35 responses

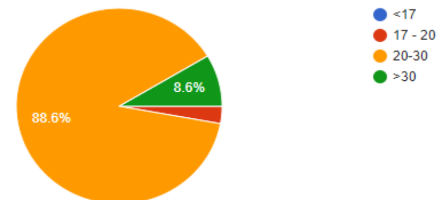

Fig. 20. Player Age Pie Chart

Figure 20 shown a pie chart of the age of players who have played the game and fill out this questionnaire. 17-20 years old has 1 player, and has total ratio is 2.9%. Age 20-30 has 31 players and has a total ratio of 88.6%. Ages over 30 have 3 players and have a total ratio by 8.6%. And the last one is under 17%, which has no respondent. The following is the calculation result from the GUESS evaluation.

After the average of each category is calculated, the average value of each category can be calculated to get the entire GUESS average as the value of player satisfaction

TABLE. 1. GUESS Average Result

GUESS subscale	Average
Usability	88.1% (Very good)
Narratives	79.6% (Good)
Play Engrossment	78.6% (Good)
Enjoyment	85.7% (Very good)
Creative Freedom	83.7% (Good)
Gratification	91.45% (Very good)
Visual Aesthetic	84.9% (Very good)
Average Result	84.58% (Very good)

From the result that can be seen in Table 1, it can be concluded that the average calculation of the category GUESS got an average rating of 84.58%, that the game has done this is very good based on the GUESS rating. The highest value is obtained from the visual aesthetic category is 91.45%, while the lowest value is obtained from the play engagement category was 78.6%.

V. CONCLUSION

Based on the research that has been done, the game with the roguelike genre using the Drunkard's Walk algorithm has been successfully implemented in the design and development of this game. This game is designed using the Unity game engine. This game provides 6 levels where every 2 levels, store level will be available. Drunkard's Walk algorithm is applied to perform generation of levels that provide a different experience. Based on the results of questionnaire that has been filled out by 35 people with different ages, the results are obtained using the GUESS assessment is 84.58%, which results very good.

Based on the research that has been done, the following are suggestions that can be given, first added difficulty system which will increase the difficulty level of this roguelike game that will add a variety of levels. Second, adding music and sound effects to the game will add audio aesthetics. Third, optimizing level creation to reduce resources to be used on the computer. Forth, increasing the attractiveness of this game so that GUESS's rating in the Play Engagement category increases. Fifth, conduct research on the Drunkard's Walk algorithm so that the implementation of the algorithm is successful.

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] W. Istiono, "Leveling up difficulty model based on user experience in education games mobile-based for student kindergartens," *IJNMT (International Journal of New Media Technology)*, vol. 7, no. 1, pp. 18–22, 2020, doi: 10.31937/ijnmt.v7i1.1666.

[2] L. Fernández-Núñez, D. Penas, J. Viteri, C. Gómez-Rodríguez, and J. Vilares, "Developing Open-Source Roguelike Games for Visually-Impaired Players by Using Low-Complexity NLP Techniques," *mdpi*, p. 10, 2020, doi: 10.3390/proceedings2020054010.

[3] H. Goandy, "No Escape: A 2D Top-Down Shooting Roguelike Game Embedded with Drunkard Walk Algorithm," *International Journal of Advanced Trends in Computer Science and Engineering*, vol. 9, no. 2, pp. 1045–1049, 2020, doi: 10.30534/ijatcse/2020/22922020.

[4] M. R. Johnson, "The Use of ASCII Graphics in Roguelikes: Aesthetic Nostalgia and Semiotic Difference," *Games and Culture*, vol. 12, no. 2, pp. 115–135, 2017, doi: 10.1177/1555412015585884.

[5] 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, doi: 10.1177/0735633117706909.

[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, [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2013/4351%5Cnhttp://drops.dagstuhl.de/opus/volltexte/2013/4336/

[7] I. G. N. Taksu Wijaya, S. Hansun, and M. Bonar Kristanda, "DISDAIN: An Auto Content Generation VR Game," *Indian Journal of Science and Technology*, vol. 12, no. 7, pp. 1–7, 2019, doi: 10.17485/ijst/2019/v12i7/141370.

[8] J. Huang and Y. Peng, "Simulation of Life Game Based on Cellular Automata," *Journal of Computer and Communications*, vol. 09, no. 01, pp. 44–58, 2021, doi: 10.4236/jcc.2021.91005.

[9] K. Vayadande, R. Pokarne, M. Phaldesai, T. Bhuruk, T. Patil, and P. Kumar, "Simulation of Conway'S Game of Life Using Cellular Automata," *International Research Journal of Engineering and Technology*, pp. 327–331, 2022, [Online]. Available: www.irjet.net

[10] R. Dogaru and L. O. Chua, "Mutations of the 'game of life': A generalized cellular automata perspective of complex adaptive systems," *International Journal of Bifurcation and Chaos in Applied Sciences and Engineering*, vol. 10, no. 8, pp. 1821–1866, 2000, doi: 10.1142/S0218127400001201.

[11] G. Oxman, S. Weiss, and Y. Be'ery, "Computational methods for Conway's Game of Life cellular automaton," *Journal of Computational Science*, vol. 5, no. 1, pp. 24–31, 2014, doi: 10.1016/j.jocs.2013.07.005.

[12] A. Dhatsuwan and M. Precharattana, "BLOCKYLAND: A Cellular Automata-Based Game to Enhance Logical Thinking," *Simulation and Gaming*, vol. 47, no. 4, pp. 445–464, 2016, doi: 10.1177/1046878116643468.

[13] P. Mouncey, L. Mlodinow, A. Lane, and P. Mouncey, "Book Review: The Drunkard's Walk – how Randomness Rules Our Lives," *International Journal of Market Research*, vol. 51, no. 5, pp. 707–708, 2009, doi: 10.2501/s147078530920089x.

[14] B. M. F. Viana and S. R. Dos Santos, "Procedural Dungeon Generation: A Survey," *Journal on Interactive Systems*, vol. 12, no. 1, pp. 83–101, 2021, doi: 10.5753/jis.2021.999.

[15] N. A. Barriga, "A Short Introduction to Procedural Content Generation Algorithms for Videogames," *International Journal on Artificial Intelligence Tools*, vol. 28, no. 2, pp. 1–11, 2019, doi: 10.1142/S0218213019300011.

[16] 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, doi: 10.1145/2422956.2422957.

[17] B. M. F. Viana and S. R. Dos Santos, "A Survey of Procedural Dungeon Generation," *Brazilian Symposium on Games and Digital Entertainment, SBGAMES*, vol. 2019-Octob, pp. 29–38, 2019, doi: 10.1109/SBGames.2019.00015.

[18] G. Ehrhardt, "The not-so-random Drunkard's walk," *Journal of Statistics Education*, vol. 21, no. 2, 2013, doi: 10.1080/10691898.2013.11889679.