Placing Sound in Procedurally Dynamic Generated Video Game Levels

Rodrigo Soares Granja
Supervisor : Louis Philippe Lopes
Master Semester Project IC - 2020
EPFL

***Abstract*— Games are complex and hard endeavours. Some can use procedural level generation to create unique content but it does not ensure an interesting experience by itself. We use sound to enhance the player experience by automatically placing and selecting audio tracks within dynamic procedurally generated levels. We use a simple PCG technique to generate the level and implement two algorithms for sound placement. The Path algorithm computes several paths and their intersections to place sounds. The Distance algorithm uses collectibles as reference points. It places sounds at multiple distances from these. Using a predefined sound library we select sounds using the Valence[9] model or player progression. Finally we propose an objective evaluation function capable of scoring the audio setup. By comparing our implementations to a baseline algorithm that is fully random we can show that our algorithms are consistent and perform well. We show that Path algorithm is more accurate but requires checking for overlaps. Distance algorithm covers more area placing more sounds but can overload the level and require tuning.**

I. Introduction

Procedural Content Generation (PCG) is the method of algorithmically generating digital game assets. Much has evolved in this field as we transitioned from simple pseudo-random functions for texture generation (such as Ken Perlin's Perlin Noise function [1]) to complete complex procedural generated universes. No Man's Sky is a recurring example in this field, it uses PCG to generate most of its content and gives a sense of discovery to players. It generates complete worlds including its terrain, weather, flora and fauna **on-the-fly**. This means players get to experience new and different worlds each time they arrive at undiscovered planets.

Nevertheless, one of the hardest aspects of PCG is how to keep the player immersed and give him the most unique, engaging and interesting experience possible where levels are generated on-the-fly and are different each time. This project focus on Sound as means to give a more interesting experience to players. We do not aim to create audio tracks procedurally but instead to automate sound placement in a meaningful manner using the game mechanics (collectibles) and level architecture (A* Pathfinding).

II. Related Work

This section explores previous endeavours in the field of procedural content generation and applications in game development.

A great example of procedural content generation applied to game development PCG are *Rogue* like games. This genre has players explore dungeons, fight enemies and collect items and experience. It makes heavy use of procedural content generation for most of its content but specially for their maps and levels.

Naturally we can find research on developing new, simpler or more efficient PCG techniques for level generation. These focus on automatically creating assets for easier game development. A few techniques are described in these papers [2][3][4]. One can see how different the approaches are.

Recent techniques using Wave Function Collapse algorithm [6] perform well and give interesting results. Nevertheless they require preparation, thus we choose Bob Nystrom's algorithm[10], for its simplicity and available code.

Others attempt to incorporate player's emotions into the procedural generation in the hopes of giving a better experience for players. In this case the players emotions are important and require consideration. The *Horror* genre and *Tension* emotion can be coupled with PCG to attempt to provide a more interesting experience for players as shown in [5].

Research on sound design and application in general games is abundant in the field. Karen Collins [7] does a good job at reviewing concepts and useful techniques. Nevertheless these are not aimed at automating placement of sound in procedural generated levels but instead to educate human sound designers. We try to fill a void in the field and focus on automatic sound placement and selection in procedurally generated levels.

III. Methodology

A. Game

1) Core Components: This project uses a collectible game in Unity with following core components:

- Procedurally Generated Level at runtime.
- First Person View player experience.
- Items to be collected.
- Optional NPC.

We chose to implement a Horror game. It is known that sound is very important for designing a horror atmosphere. Bernard Perron in his book[8] gives a critical and theoretical analysis of the horror genre in video games and approaches the sound aspect in the genre. It is a great source of inspiration for us as it allows us to have a more in-depth understanding of how sound is utilised to create a compelling horror experience.

2) Player: The game uses a First Person View player represented by a red cylinder. Players can jump and run at constant speed around the maze. They cannot defend themselves from the NPC but run faster than it. Players can also interact with collectibles to retrieve them.

3) Optional NPC: We call it the maze spirit and it is a red capsule. It represents the danger in the level. The spirit roams slowly around the maze going from room to room and if he sees the player he starts to glow orange, speeds up and tries to consume the player. After losing sight of the player it eventually stops chasing and resumes roaming.

By default the NPC is disabled since it is much complex to analyse the sound placement with this feature. Code concerning the NPC can be found at the "npcBehaviour" script.

4) Game Objective: Players must explore the maze and ultimately find 4 collectibles. These are called "Totems" and are scattered around the level. They are represented by magenta emitting cubes floating at the center of rooms. These locations are unknown to the player.

Game ends with the player winning if he collects all totems. Player loses if the spirit reaches him. Please notice that if the optional NPC is not enabled the player cannot lose.

5) Unity interface: Users can use the "Game Controller" component for settings before running the game and tracking the player progress.

For a better user experience and to avoid issues we wrote a custom Unity editor for this component. It adapts to the user choices and disables certain options when the game is running.

We list some of the most important options of this component :

- Player speed. The higher this setting is the faster the player will move.

- Sound algorithms, users can choose among 3 algorithms. These will be explained later on.
  - Baseline Algorithm
  - Path Algorithm
  - Distance Algorithm

- Enable playthrough simulation
  Only if enabled will all simulation options become available to the user. Such as setting the criteria weights and the number of simulations to perform.

- Set the number of collectibles. This option is disabled after the game starts.
  Please notice that our implementation is optimized for 4 collectibles.

- Load and Save options.
  - Load Level. Users can choose to load both level and collectibles or only the level, generating new random positions for collectibles. Load options are disabled after the game starts.
  - Save Level. This checkbox only appears while the game is running and allows the player to save the current level and collectible positions.

    If the player chooses to save the game, files will be overwritten directly at the destination. The load/save path is at : ProjectRoot\Level\Rooms. If simulation is active the evaluation function results will be saved at ProjectRoot\Level\Eval.

- Enable NPC. Players can not enable NPC if the game is running.

B. Level Generation

One of the main requirements of this project is the Procedural Content Generation aspect of the game level. Levels must be created on-the-fly and thus are unknown before runtime.

1) Bob Nystrom Algorithm: While there are many well known PCG Dungeon algorithms we choose to use Bob Nystrom's algorithm [10] which implements a maze level filled with rooms, doors and corridors. We choose this algorithm specifically because it creates rooms that are useful for placing collectibles and it is fully connected with no dead-ends.

The algorithm starts by randomly placing rooms with different sizes. It proceeds to grow a maze around the rooms and then connecting them by adding doors. Finally it removes all dead-ends.

David Schwarz has a Unity implementation of this algorithm [11] based on Unity's Tilemap functionality for creating 2D levels. We adapt his code to create a 3D with our own prefabs and materials. Each element of the scene has corresponding material according to what it corresponds to : roof, floor, wall, door. These can be found under "Assets/Graphics".

"Nystrom Generator" script contains every data structure related to the level. It contains the following structures : rooms including collectible rooms and starting room, important paths, important intersections following those paths.

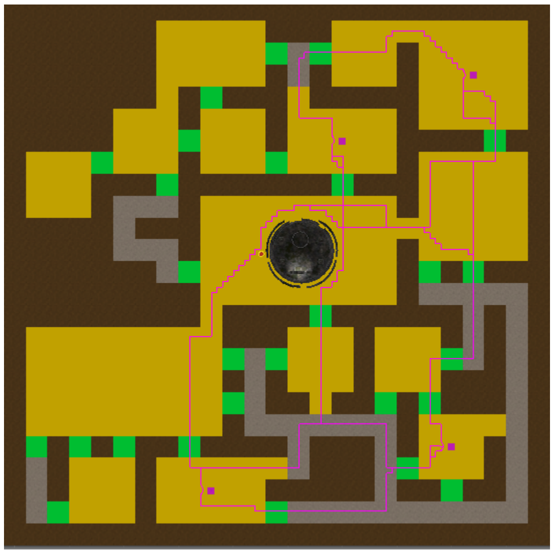


Fig. 1: Top view of our level. We use these colors for visual simplification. Yellow : room floor, Green : doors, Brown : walls and Grey : corridors. Smaller magenta points are our Totems, red capsule is our player and the big structure in the bottom left room is a mesh that is placed inside the starting room representing a ruined altar.

2) A Star Algorithm: In our implementation we require calculating paths from point A to point B. We use these for :

- Use in algorithms
- Verifying correctness of the level after creation. We check if the player can reach every collectible.
- Simulating playthroughs.
- NPC behaviour.

The A* search algorithm is a very well known path search algorithm. We use the A Star Project framework [13] since it is well established with good documentation and performance in Unity.

It works in 2 steps, first we create a game object "AStar" containing the "Astar Path" script. This is the base of the algorithm as it contains node graphs. Nodes can be walkable or not and paths use them. The script uses layers for setting nodes access, and as such we must set which layers are obstacles by setting the "Collision Testing" toggle and defining such layers. In our implementation we set the following layers as obstacles : Walls, Collectibles and Obstacles. Nodes that are covered by these layers become inaccessible. Our logic is that players or NPCs must not be able to walk over walls, collectibles or other obstacles. The second step is to look for paths. The "Seeker" component allows us to do this. In our implementation, our "Nystrom Generator" script uses a "Seeker" component for calculating important paths. It finds the nearest Astar graph nodes to the starting and end positions respectively and attempts to find a path. If one of such paths is impossible then the game is impossible to complete, so we restart it entirely.

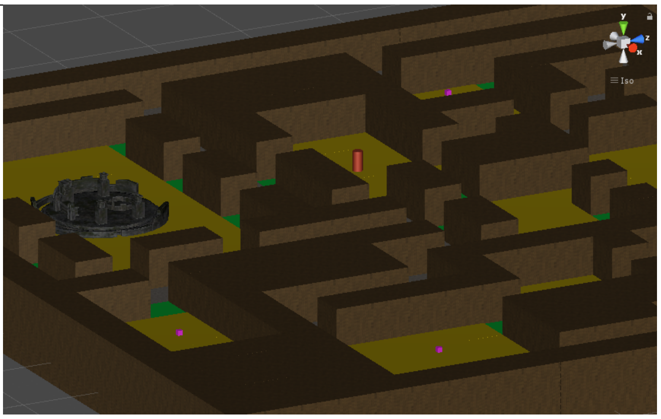


Fig. 2: In this image we can see a 3D view of our level made of cubes. We can also see the player (red capsule) and collectibles (magenta cubes) more easily. On the left there is our altar structure representing the starting room.

The path calculation is done within the "Nystrom Generator" script.

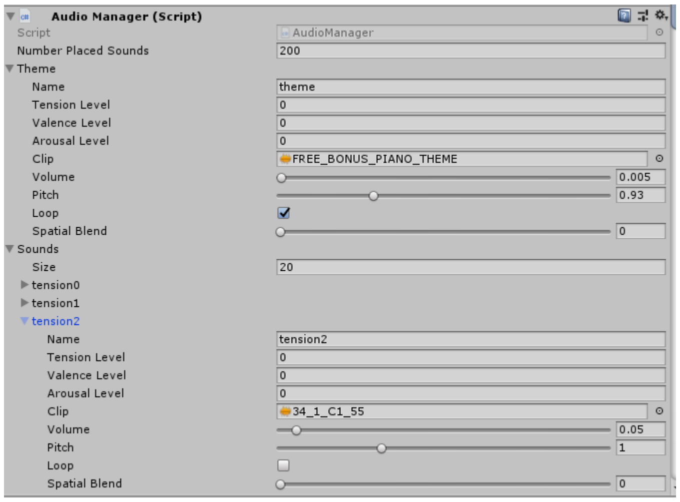
IV. Sound

We are aware that one important aspect of the horror sound experience is contrasting diegetic (source in-game : e.g. voices) and non-diegetic (outside story space) sounds but to somewhat limit the scope of this project we decided to solely focus on non-diegetic sound tracks putting more emphasis on the level atmosphere. The objective of this project is not to dynamically create the sound tracks, therefore we used an already existing sound dataset created by Lopes et al. [12] called Sonancia. We classify sound tracks based on the "Arousal-Valence" model [9] and this sound dataset contains preference files for "Tension", "Arousal" and "Valence" that we can use. We create a script that uses those files and gives an ordered list of sounds by classification attribute. In our implementation we will focus only on "Tension" and "Valence". "Tension" defines how stressful audio is. "Valence" classifies audio by ranking it from pleasant (high valence) to least pleasant emotions triggered (low valence)

Finally we propose a multi-objective evaluation function with a few criteria. This function computes a score of our implementation after a playthrough simulation. It can be used to verify the performance of our algorithms.

A. Sound setup

To make the use of the sound library as simple as possible we decided to implement a small interface through the means of our ”AudioManager” game object. Users can set the theme sound, which will directly play at the start of the game, and also construct the sound database by manually adding each sound. Users can see that there are a few options available when setting the audio tracks, such as volume, pitch, loop and audio blend. Finally it asks for the max number of sounds to be placed in the level.



```
Audio Manager (Script)
Script                    AudioManager
Number Placed Sounds      200
▼ Theme
    Name                  theme
    Tension Level         0
    Valence Level         0
    Arousal Level         0
    Clip                  FREE_BONUS_PIANO_THEME
    Volume                0.005
    Pitch                 0.93
    Loop                  ✓
    Spatial Blend         0
▼ Sounds
    Size                  20
  ▶ tension0
  ▶ tension1
  ▼ tension2
      Name                tension2
      Tension Level       0
      Valence Level       0
      Arousal Level       0
      Clip                34_1_C1_55
      Volume              0.05
      Pitch               1
      Loop                ☐
      Spatial Blend       0
```

Fig. 3: Unity editor view of the ”Audio Manager” component. First we see that we can set the maximum number of sounds to be placed. Then we manually add sounds to both Theme and Sounds containers. Several sound options are also available.

We now go more in-depth of what happens once the sound database is manually set.

We created a class ”Sound” for representing our audio tracks. It contains all the options seen in the editor when creating the sound database. Unity uses *AudioClip* objects for representing an audio file. *AudioSource* are components used to represent said audio files in the game world. In our implementation, the user sets the AudioClip that is meant to be played using the Unity editor, and our implementation in ”AudioManager” iterates over these sounds and creates the *AudioSource* components. This has an important consequence in our implementation, each sound has a unique sound source. This means that if we place two equal sounds, **we can only play one of these at a time** . We provide a wrapper method for playing the audio clips, e.g. ”PlayOneShot()” definition in ”Sound” script.

Furthermore we wrote a ”SoundBehaviour” script. It is in charge of defining when sounds should trigger. We designed our sounds as to trigger only in vicinity of the player.

B. Sound Evaluation

1) Game Simulation: The evaluation is a way to check how our algorithms perform but we can only know this after running the game and simulating a playthrough. This is why we implement a simple AI that plays the game for us. Keep in mind that the NPC should not be enabled simultaneously with the AI agent.

Script ”AIAgent” contains all code relative to this section. The AI algorithm we implement is as follows :

- Set destination to a random room
- While the AI is going to the room we check for collectibles. That is, we check if there are collectibles directly visible by the AI.
  - If there is, we store the room the AI was currently going.
  - Set the collectible as new destination.
  - Resume previous destination after collecting the item.
- If the AI agent enters a room it marks it as visited and no longer needs to return there.
- Simulation ends after collecting all 4 totems.

It uses the A Star Project framework for automating movement of the player. For this we require a *AIPath* and *AIDestinationSetter* components. these combined allow the player to move automatically be setting the *target* value of the AIDestinationSetter component to the room centers. We apply a path smoother (*SimpleSmoothModifier* component) so the AI agent movement is more believable).

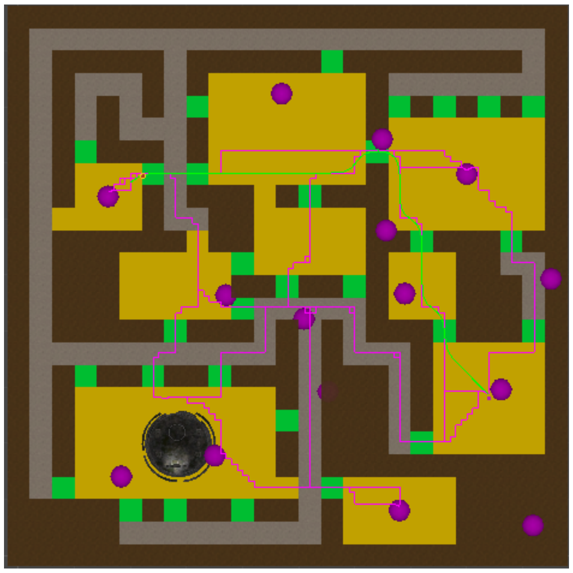


Fig. 4: Figure displaying the path the AI agent is currently taking in green. We can easily notice how the path is smoother and different from previous calculated paths.

2) Criteria and weights: Once the simulation ends, e.g. all 4 totems are collected, we run the evaluation function. This function is implemented in the ”AIAgent” script.

We take interest in sound elements that can give an objective quantification of the performance of our algorithms. Each criterion score ranges from 0 to 1, finally we compute the total score averaging over all criteria taking their weights into consideration. Thus, by construction total score ranges from 0 to 1.

Some criteria may be less pertinent for a specific algorithm or people may give more importance to some criteria. We give the possibility to users to assign weight values to criteria, thus allowing for tuning the evaluation function. These weights can be defined in the ”Game Controller” component through the Unity editor if Simulation is toggled.

Here we present the criteria we use to evaluate performance.

- Repetitive Criterion : Are sounds being repeated?
  As in any experience, hearing the same audio track multiple times in one playthrough diminishes the immersion felt by the player. This criterion assigns scores to each played sound and computes the average over all played sounds. Individual scores are calculated according to the following play count :
  - 1 time : + max score
  - 2 times : + half score
  - 3 times : + 1/4 score
  - more than 3 times : - half score

- Count Criterion : Is the ratio of played/placed sounds good enough?
  This criterion evaluates if the sounds placed were triggered, and gives a score based on this. We average sound locations triggered over all sound locations to compute the score. For a perfect score, all placed sounds should be triggered.

- Path Coverage Criterion : Do we have good path coverage?
  This criterion verifies how much level surface is covered through paths. We check which path nodes are covered and average over all path nodes. If at least half of all path nodes are covered we give max score. Otherwise we linearly map values from domain 0-0.5 to 0-1. This is done because it is impossible to achieve full path coverage since we use our refinement function to avoid close sound placements.

- Room Coverage Criterion : Does the player listen to sounds in rooms?
  Here we check the level coverage by checking if rooms are covered. If 2/3 of all rooms are covered we assign max score, otherwise we linearly map values from domain 0-0.75 to 0-1.

- Progression Criterion : Are the sounds being played coherently with the progression of the player, e.g. number of collected totems.
  We realise this criterion is directly handled by the first sound selection algorithm (Path algorithm), Nevertheless it is also a good way to evaluate the baseline and distance algorithms.

- Overlap Criterion : Are sounds overlapping too much?
  When sounds are too close they tend to overlap. Having too many sounds playing over one another is detrimental to the player experience. We give max score if no sounds overlap and linearly decrease the score until the number of overlaps equals or is greater than the number of sounds placed.

C. Sound Algorithms

We start by presenting a baseline algorithm for sound placement and selection which is random within the bounds of the level. Then we analyse 2 (two) algorithms for sound placement. The first, Path Algorithm, uses A* Pathfinding algorithm to search for important paths in the level and compute their intersections. We use the A Star Project framework [13] since it is very well established with very good documentation and performance in Unity. This algorithm selects sounds based on the level progression, e.g. how many collectibles the player retrieved, and updates the sounds accordingly. The second, Distance Algorithm, uses the collectibles' locations as circle center points with multiple radius, then computes circle intersections. The second algorithm takes into consideration the distance to collectibles to pick sounds with more or less valence.

All algorithms are implemented in the ”AudioManager” script.

Baseline Algorithm

This algorithm randomly chooses and places sounds in the level. It is meant to be compared to the other more complex algorithms. We select a random sound id from the sound container and place it randomly within the bounds of the game level. This algorithm places the exact amount of sounds that is defined in the script. This value may be modified in Unity editor under the ”AudioManager” component script.

Path Algorithm

This algorithm makes use of the A Star implementation to calculate important paths and their intersections.

First we calculate all meaningful paths, suchs as:

- Player to all collectibles.
- All paths from collectible to collectible.

This allows us to have paths that will most likely be taken by the player. These paths contain a list of walkable node positions and by storing and comparing all paths we can find points of interest translated by their intersections. By itself this is not enough so we also check if paths intersect with doors and add these to our potential sound positions.

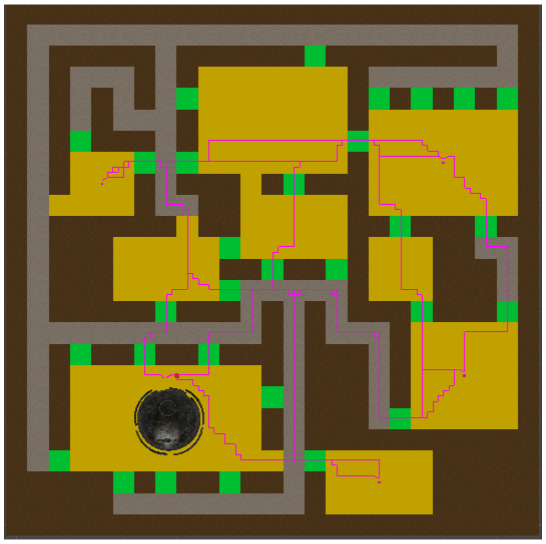


Fig. 5: Same level with important paths calculated with the A Star Project framework highlighted.

To avoid issues with overlapping sound positions we implement a refinement function. Our simple AStar node graph computes straight vectors from node to node. Thus paths are not smooth, as one can see in figure 5. To correct this problem we implement a ”RefineSoundSpots()” function that checks if two intersections are close enough, and in that case replaces both intersections by the new average position. Users can define how many times this function is called (Refine Iterations) and the distance threshold (Sound Closeness) by setting the corresponding settings in the Unity editor at the ”NystromGenerator” component in the ”LevelGenerator” scene object.

Images 6 and 7 illustrate how this function changes the sound placements.

Distance Algorithm

One issue with the first algorithm is that most sound positions are close to the collectibles. . This is expected since the algorithm uses close paths to each collectibles. Nevertheless, it also means that players will rarely hear sounds when far from these points. Thus, this second algorithm attempts to address this issue.

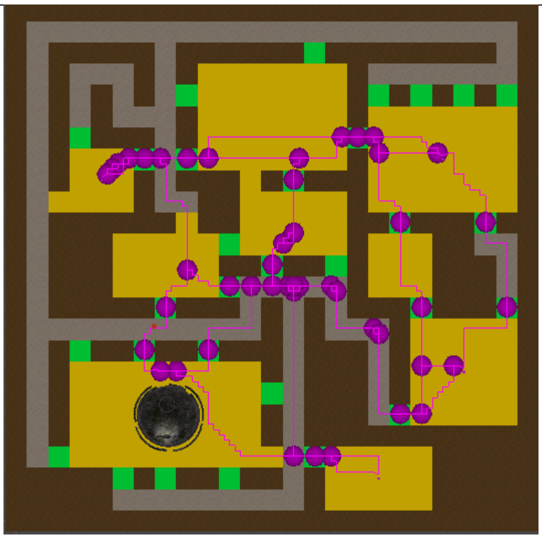


Fig. 6: Algorithm 1. Sound placements before refinement. Notice how close spots are. One can see how paths are not smooth and thus provoke these intersections.

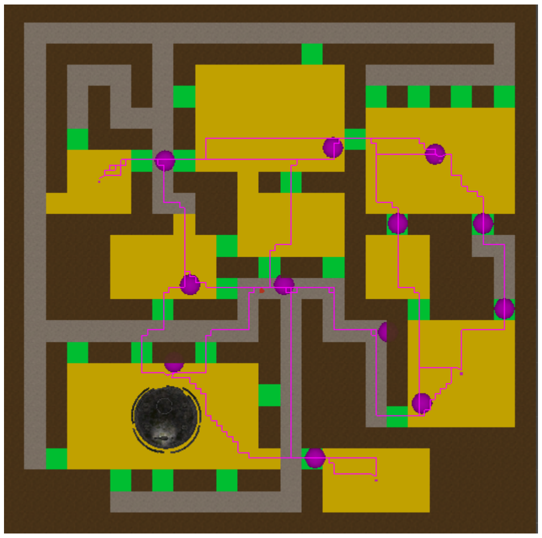


Fig. 7: Algorithm 1. Sound placements after refinement. Sound spots are more spaced between them.

We focus on distance to collectibles. We start placing sounds at every collectible position. These positions become the center of 4 circles with 4 different radiuses. The shortest radius equals the level dimension / 8 and the longest equals half the level dimension (e.g. level width). Shorter radius imply closer distances to collectibles. Figure 8 displays how these circles appear in the level. Then we compute intersections between pairs of circles with same radius and place sounds at these positions.

Figures 8, 9 and 10 show our Distance algorithm in practice

One can observe that circles with the highest radius will most likely intersect and thus we will be placing sounds far from collectibles.

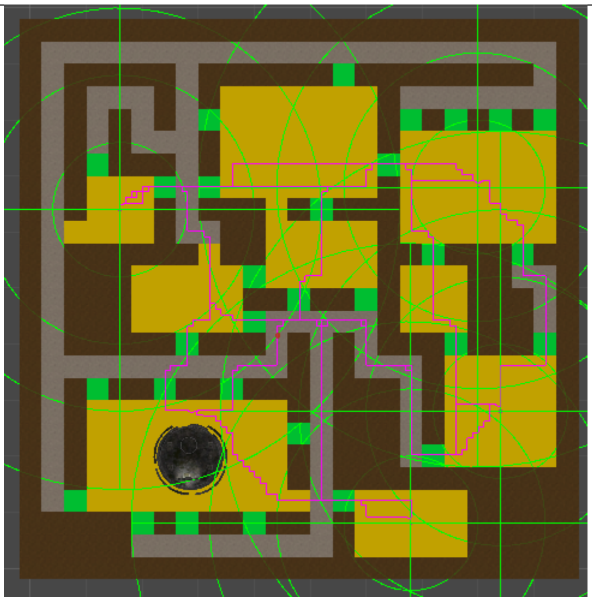


Fig. 8: Start of algorithm 2, defining multiple circles centered at each collectible location with different radius.

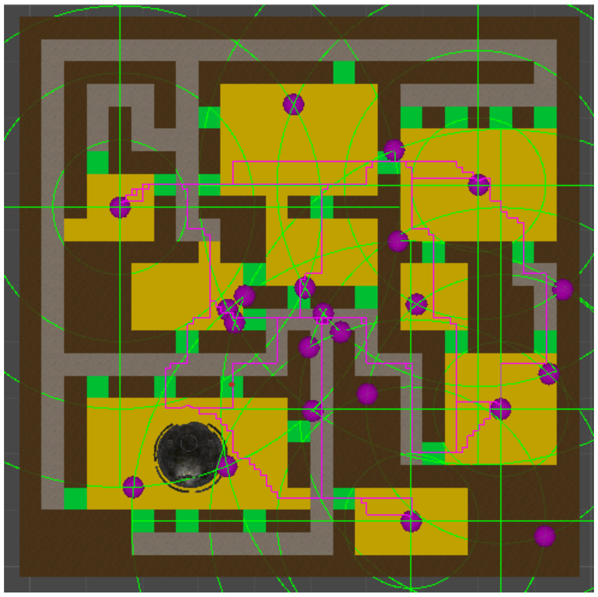


Fig. 9: We find intersections of circles with same radius and add these positions to potential sound positions. Here we can see how close these positions can be, before refinement.

In this algorithm we also use the same refinement function to avoid close and overlapping sounds. We slightly adapt the refinement such that for this algorithm we still end up with sounds very close to the collectibles' location.

1) Sound Selection: We have 2 sound selection implementations. It is important to notice that these rely on *Valence* and *Tension* levels, and these must be set up by the user using the "AudioManager" component and its *Valence Level* / *Tension Level* variables for each sound in the database before running the game. See Figure 3.

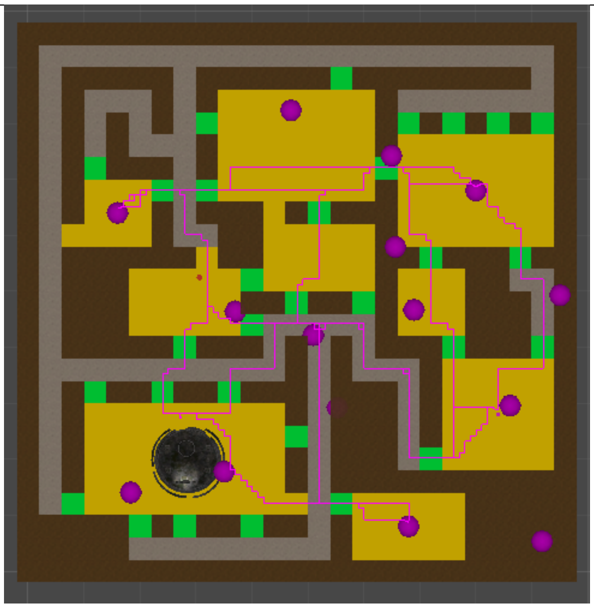


Fig. 10: At the end of algorithm 2 we can apply the refinement function to space out sound positions to achieve this final result. We can observe that there are more positions that seem to cover more area of the level.

- The first implementation is dynamic, tracks player progression and uses sound Tension classification. This is implemented for our Path algorithm .

  At the start of our selection implementation, we define the current progression level. This is derived from the number of collectibles that the player has retrieved. There are 4 collectibles so progression ranges from 1 to 4. 1 when the player starts the game, all the way to 4 when the player is missing one collectible. Progression level is 1 at the start of the game, so we find all sound tracks with tension level equal to 1 and randomly choose among them and place them.

  Once players find a totem and collect it, we update all sounds. We start by destroying all previously placed audio. We do this by finding all Game Objects with tag "Sound". Then we call our selection implementation algorithm again, calculating the new progression level from the number of collected totems, selecting sounds with appropriate tension level and placing them in the scene. Placement is done using the Path algorithm.

  With this approach sounds will be modified according to player behaviour and hopefully the player will feel he is progressing thanks to the changing audio tracks.

- The second implementation uses sound Valence classification. We imagine our totems as sources of negativity. Thus, sounds close to them should have lower valence triggering less pleasant emotions. Further away locations should have higher valence. We use this to select which sounds to place **after** calculating the sound positions. This selection implementation is used alongside the Distance algorithm.

For each placed sound we compute the distance to the closest collectible. This distance is used to compute the expected Valence level of this sound. We define 4 different threshold distance **intervals** and match the calculated distance to these intervals. The intervals depend on the level dimensions.
Valence Levels by interval:

1) 0 to level_width / 8
2) level_width / 8 to level_width / 4
3) level_width / 4 to level_width * 3 / 8
4) level_width * 3 / 8 to level_width / 2

With this implementation sounds also become useful. Players can interpret these to guide themselves throughout the maze. Less pleasant audio hints that there are nearby collectibles. It also gives the feeling that locations have meaning, giving life to the environment.

V. Experiments

Results are separated in 2 sections. We use the "Nystrom Generator" component to modify the level dimensions. We define the level dimension to be the width of the level. This value may be modified using the "Nystrom Generator" component at the corresponding field.

First section will compare our algorithms on 2 different levels with small and big dimensions, 25 and 51 respectively. Levels can be seen on figures 11 and 12. This allows us to see how our algorithms differ on the same level architecture. Second section will compare our algorithms on 20 simulations with standard dimension of 29. Levels will be randomized at each simulation. This allows us to check how robust our algorithms are.

We do not go into detail over the repetition criterion since it is highly affected by the size of the sound library. In our case we use a library of 22 sounds. Since we place around 10-20 sounds at each simulation, there are great chances of sounds being picked repeatedly. One can augment the size of the sound library to avoid repetition.

The Path Algorithm consistently has max score at the Progression criterion since it is set as one of its rules. For this reason we ignore this criterion in our results.

1) Results for constant architecture:

Small Level dimension - 25

We simulate 20 playthroughs for each algorithm. In all 3 cases algorithms place 12 sounds in the level. Figures 13, 14 and 15 contain box plots of our results. These contain all criteria and total score.

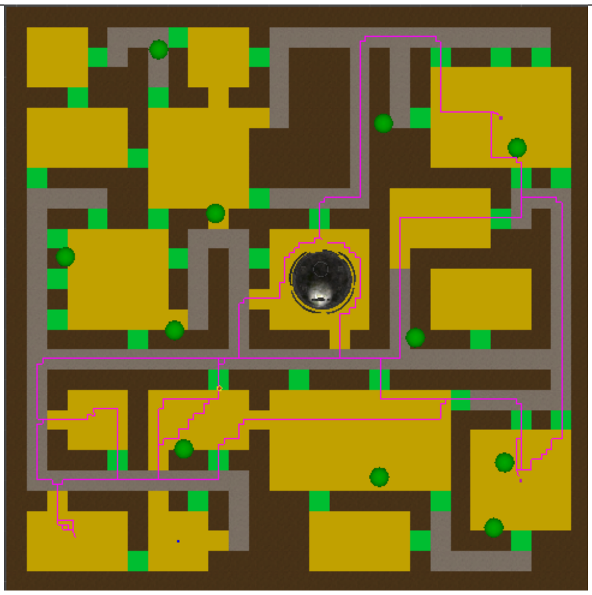


Fig. 11: Level of dimension 25 used for simulations. It shows one example of the Baseline Algorithm placements and important paths.

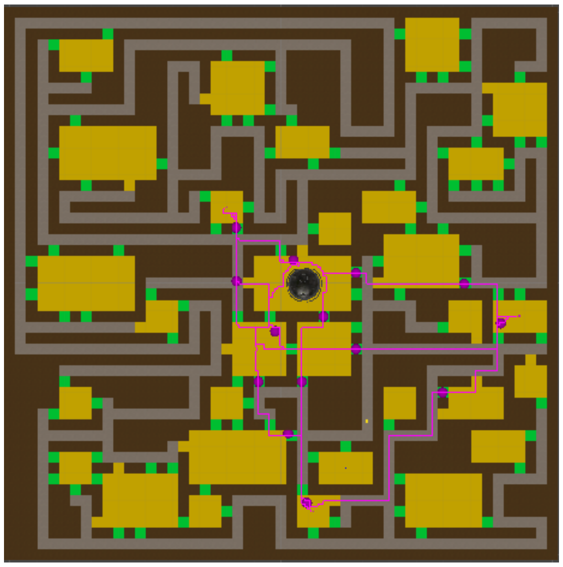


Fig. 12: Level of dimension 51 used for simulations. It shows placements based off the Path Algorithm.

We can observe how total scores are similar between our 2 implementations when level is small and perform better than the baseline algorithm. Nevertheless Distance Algorithm performs better and consistently gives better results. One can observe the following differences :

- Path Algorithm has better evaluations in Count and Path Coverage criteria. Very bad evaluations for Room Coverage and Overlap criteria.

  This is expected since this algorithm follows paths

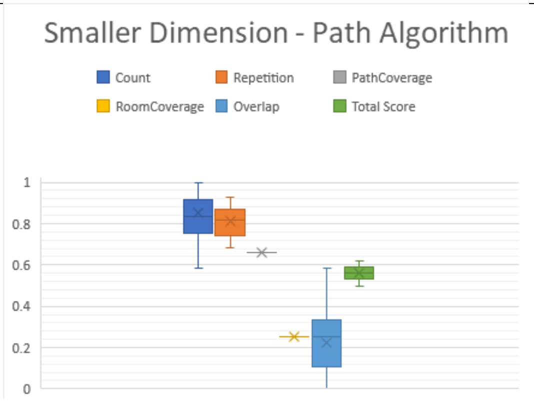


Fig. 13: Box Plot graph of our Path Algorithm performance for 20 simulations on the same smaller dimension (25) level .

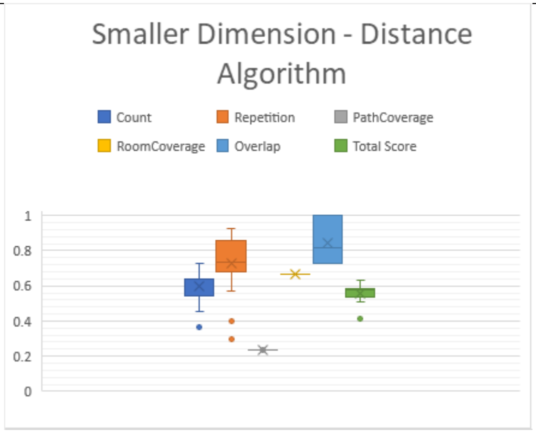


Fig. 14: Box Plot graph of our Distance Algorithm performance for 20 simulations on the same smaller dimension (25) level .

which are more likely to be taken by the player. This means sounds are more likely to trigger and will be closer to paths. Our sound selection algorithm makes sure that sounds are updated according to progression, thus sounds are less likely to be repeated. It is also expected that this algorithm fails at Room coverage and Overlap criteria because sounds tend to be more concentrated on the collectibles area. Thus, sounds are close to each other and overlap and do not cover a big level area.

- Distance Algorithm has better evaluations generally, thus having a better total score overall in the small level. One can observe how the Overlap criterion is significantly better, since it places sound further away from each other. As expected it has better room coverage evaluation which it was designed to achieve.

- We can compare our algorithms to the baseline algorithm which uses randomness. We can observe how our algorithms are much more consistent accross all criteria, thus having more consistent Total Scores. Our box plots demonstrate how they perform better overall than the baseline algorithm.

Greater Level dimension - 51

When we increase level dimensions we also increase number of rooms, path lengths and level area. This allows us to analyse how our algorithms scale with level sizes. For this result section we simulate 10 playthroughs for each algorithm on the bigger level with dimension 51.

By construction, the Distance Algorithm can potentially find many more sound placements than the Path Algorithm. To avoid that the number of sounds in the scene influences the results we limit the max number of sound placements for the Distance Algorithm. For this level the Path algorithm places 16 sounds. Thus, we set the max number of sounds to be placed using the Distance algorithm to this value.

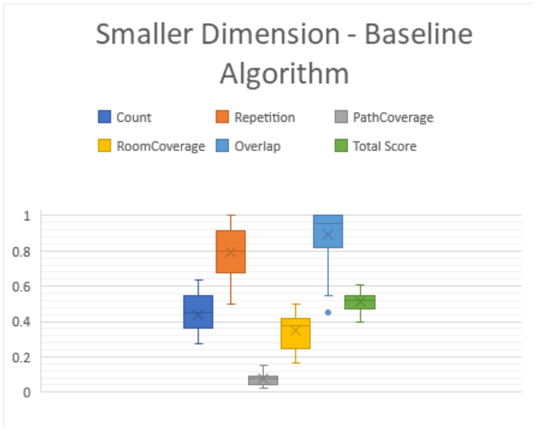


Fig. 15: Box Plot graph of our Baseline algorithm performance for 20 simulations on the same smaller dimension (25) level .

Our results in figures 16 and 18 when compared to figures 13 and 14 show that scores diminish when increasing the level dimension. This is associated with the increase of the area and room number as we can see in the graphs. In the bigger level sounds are more easily missed and there are less rooms covered. We can see this from values of our box plots of Count and Room Coverage which considerably drop. On the other side we have less overlaps, and thus better score also due to increasing level area and triggering less tracks.

With the same amount of sounds placed for this level our results show that the Path algorithm scales better than the Distance algorithm. We can see consistent less performance from the Distance algorithm by checking the total score box plot. Nevertheless, one may think that not limiting the number of sounds makes the algorithm perform better. We show that allowing the algorithm to place as many sounds as it finds gives worse performance. The box plot at figure 17 translates this. One can see that the total score is significantly lower. In fact, while it places more sounds and covers more area, ratio of sounds triggered and sounds

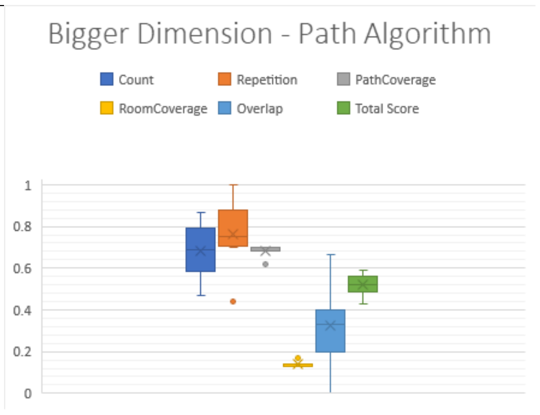


Fig. 16: Box Plot graph of our Path algorithm performance for 10 simulations on the same bigger dimension (51) level .

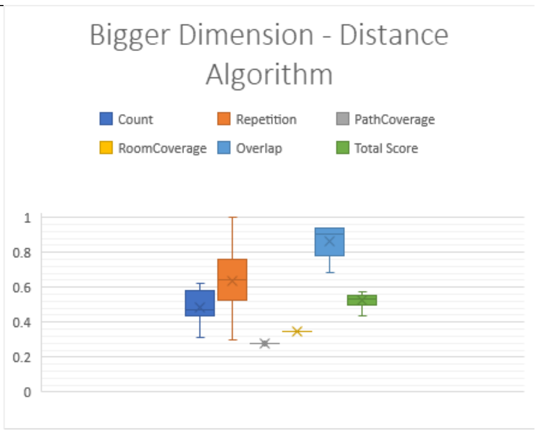


Fig. 18: Box Plot graph of our Distance algorithm performance for 10 simulations on the same bigger dimension (51) level .

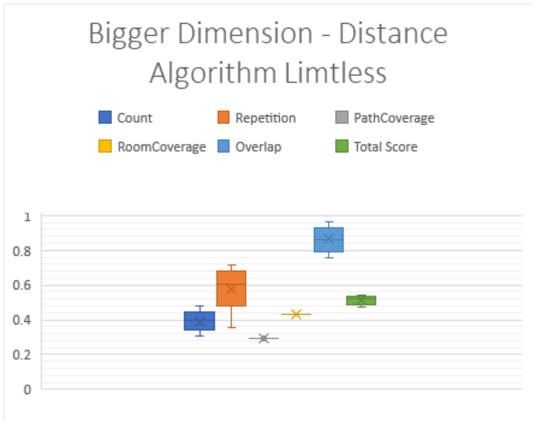


Fig. 17: Box Plot graph of the Distance algorithm when allowed to place as many sounds as it finds. Results after 20 simulations on the bigger dimension (51) level .

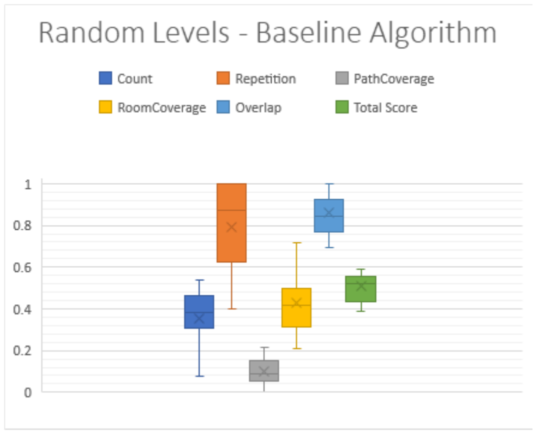


Fig. 19: Box Plot graph of our baseline algorithm performance after 20 simulations with different levels of dimension 29.

placed becomes bad. It overcharges the level with no significant benefit for players. We can see how the Count criteria takes a heavy hit while coverage (Path and Room) and Overlap criteria only slightly increase. This highlights the importance of choosing the appropriate maximal number of sounds to be placed when using the Distance algorithm.

Thus these results confirm that the Path algorithm scales better than the Distance Algorithm.

2) *Random Levels Results:*

The previous section has levels with fixed architecture and collectible locations which highly influence the Path and Distance algorithms. To remove this bias we show results accross 20 levels with same dimension. Levels are generated at runtime at the start of each simulation and thus are different every time. We chose 29 to be the levels' dimension which is a value between the small and big levels used in the previous section.

Figures 19, 20 and 21 display box plot graphs of the simulations. We find that 50% of total score values of our algorithms are around 0.6 while the baseline varies more and has values around 0.55. This shows that our algorithms have better performance and are more robust than the baseline algorithm when applied to multiple different levels.

Figures 20 and 21 show the strengths of each algorithm. Path algorithm is better at accurately placing sounds. The high Count and Path coverage criteria scores account for this. While Distance algorithm is better at covering the level area. This can be observed from the higher Room Coverage and Overlap criteria scores.

VI. Conclusion

We presented two algorithms for automatic sound placement in dynamic procedurally generated levels with two different approaches for sound selection.

Path algorithm does a good job at accurately placing sounds near important locations with good level coherence. Players hear most of the sounds that are placed and feel they are progressing when collecting items. Nevertheless,

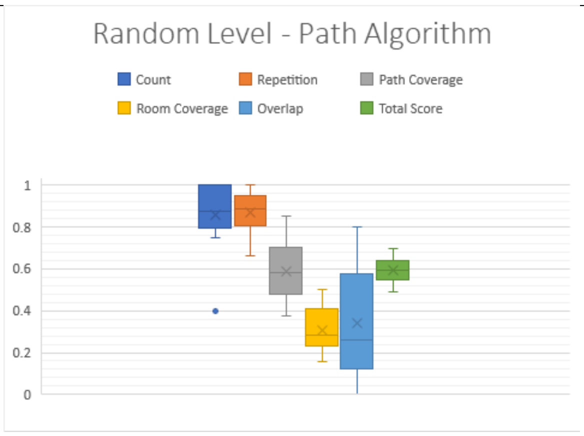


Fig. 20: Box Plot graph of our path algorithm performance after 20 simulations with different levels of dimension 29.

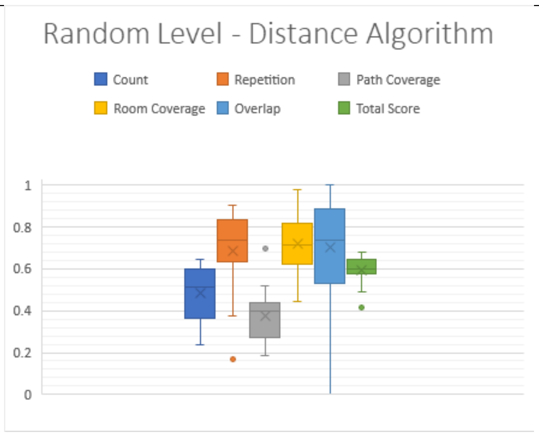


Fig. 21: Box Plot graph of our distance algorithm performance after 20 simulations with different levels of dimension 29.

it concentrates audio in these regions. This may cause players to experience long periods of silence when far from important rooms. Games may require rules in sound mechanics as to avoid audio overlapping. This algorithm scales well with level size.

Distance algorithm excels at equally covering the level surface with sounds. These are well spaced between them avoiding overlaps. It places sounds according to Valence with audio closer to collectibles being less pleasant. This makes the game environment feel alive where players can feel it changing through sound. However it does not place audio sources accurately and these might be missed. It compensates by placing a higher number of sources than the Path Algorithm. Some limitation is required as to avoid overcharging the level with sounds specially when applied to higher sized levels.

Finally we can use the objective function to evaluate algorithms performance. This function focus mainly on physical objective criteria to give a score to the implementations. We used it to show how these implementations perform consistently accross different level layouts.

Procedural content generation is an evolving field. While there is research on how to create sound dynamically, it is lacking research on how to better utilise Sound within dynamic levels to create more interesting experiences. This work shows it is possible to enhance the player experience within dynamic procedural levels by using sound. For an optimal experience the sound library used by these algorithms should be constructed to fit into the game design.

VII. Future Work

The algorithms presented approach the sound placement problem differently and each has strengths of their own. Joining both algorithms could potentially lead to better results. An interesting approach to the *sound placement* would be to use both algorithms in sequence to place sounds in the scene. Proceed by using the refinement function mentioned in this report to space out the positions. *Sound selection* algorithms could also be joined. First one would select sounds according to progression and then filter by Valence according to how far it is from the nearest collectible.

In this project we provide means to evaluate the performance of algorithms through our Evaluation Function. Implementing a genetic algorithm would be an interesting way to utilise the algorithms to automatically improve placement and selection based off this function results.

The presented evaluation function concentrates on objective physical criteria such as the coverage, overlap and repetition. These focus mainly on sound placement more than selection. It is a challenge to develop this function towards sound characteristics to better evaluate sound selection and how people experience the level.

Implementation can be found at :
https://github.com/Rudra92/TotemDungeon

References

[1] Ken Perlin, Fabrice Neyret. Flow Noise. 28th International Conference on Computer Graphics and Interactive Techniques (Technical Sketches and Applications), Aug 2001, Los Angeles, United States. SIGGRAPH, Technical Sketches and Applications, pp.187, 2001. (inra-00537499)

[2] R. van der Linden, R. Lopes and R. Bidarra, ”Procedural Generation of Dungeons,” in IEEE Transactions on Computational Intelligence and AI in Games, vol. 6, no. 1, pp. 78-89, March 2014, doi: 10.1109/TCIAIG.2013.2290371.

[3] Lawrence Johnson, Georgios N. Yannakakis, and Julian Togelius. 2010. Cellular automata for real-time generation of infinite cave levels. In Proceedings of the 2010 Workshop on Procedural Content Generation in Games (PCGames ’10). Association for Computing Machinery, New York, NY, USA, Article 10, 1–4. DOI:https://doi.org/10.1145/1814256.1814266

[4] J. Togelius, G. N. Yannakakis, K. O. Stanley and C. Browne, ”Search-Based Procedural Content Generation: A Taxonomy and Survey,” in IEEE Transactions on Computational Intelligence and AI in Games, vol. 3, no. 3, pp. 172-186, Sept. 2011, doi: 10.1109/TCIAIG.2011.2148116.

[5] Lopes, Liapis, N.Yannakakis, ”Targeting Horror via Level and Soundscape Generation”, in Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-15).

[6] H. Kim, S. Lee, H. Lee, T. Hahn and S. Kang, ”Automatic Generation of Game Content using a Graph-based Wave Function Collapse Algorithm,” 2019 IEEE Conference on Games (CoG), London, United Kingdom, 2019, pp. 1-4, doi: 10.1109/CIG.2019.8848019.

[7] Karen Collins (2009) An Introduction to Procedural Music in Video Games, Contemporary Music Review, 28:1, 5-15, DOI: 10.1080/07494460802663983

[8] Bernard Perron, Horror Video Games: Essays on the Fusion of Fear and Play, 2014.

[9] Russell, J. A. (1980). A circumplex model of affect. Journal of Personality and Social Psychology, 39(6), 1161–1178. https://doi.org/10.1037/h0077714

[10] Robert (Bob) Nystrom, https://journal.stuffwithstuff.com/2014/12/21/rooms-and-mazes/

[11] David Schwarz, https://github.com/Zeraphil/NystromGenerator/

[12] P. Lopes, A. Liapis, and G. N. Yannakakis, “Sonancia: Sonification of Procedurally Generated Game Levels,” in Proceedings of the ICCC workshop on Computational Creativity & Games, 2015.

[13] A* Project in Unity, https://arongranberg.com/astar/