Generative Adversarial Network Rooms in Generative Graph Grammar Dungeons for *The Legend of Zelda*

Jake Gutierrez[1] and Jacob Schrum[2]

***Abstract*— Generative Adversarial Networks (GANs) have demonstrated their ability to learn patterns in data and produce new exemplars similar to, but different from, their training set in several domains, including video games. However, GANs have a fixed output size, so creating levels of arbitrary size for a dungeon crawling game is difficult. GANs also have trouble encoding semantic requirements that make levels interesting and playable. This paper combines a GAN approach to generating individual rooms with a graph grammar approach to combining rooms into a dungeon. The GAN captures design principles of individual rooms, but the graph grammar organizes rooms into a global layout with a sequence of obstacles determined by a designer. Room data from *The Legend of Zelda* is used to train the GAN. This approach is validated by a user study, showing that GAN dungeons are as enjoyable to play as a level from the original game, and levels generated with a graph grammar alone. However, GAN dungeons have rooms considered more complex, and plain graph grammar's dungeons are considered least complex and challenging. Only the GAN approach creates an extensive supply of both layouts and rooms, where rooms span across the spectrum of those seen in the training set to new creations merging design principles from multiple rooms.**

I. Introduction

Video game developers increase replayability and reduce costs using Procedural Content Generation (PCG [1]). Instead of experiencing the game once, players see new variations on every playthrough. This concept was introduced in *Rogue* (1980), which procedurally generates new dungeons on every play. PCG is also applied to modern games like *Minecraft* (2009), where users play on generated landscapes, and *No Man's Sky* (2016), where procedurally generated worlds contain procedurally generated animals. PCG encourages increased exploration and increases replayability.

An emerging PCG technique is Generative Adversarial Networks (GANs [2]) used to search the latent design space of video game levels, as has been done in *Super Mario Bros.* [3], *Doom* [4], an educational game [5], and the General Video Game AI (GVG-AI [6]) adaptation of *The Legend of Zelda* [7]. In the GVG-AI version of Zelda, single-room levels require the player to fight enemies, reach a key, and take it to the exit. The technique applied by Torrado et al. [7] to this game focuses on modeling non-local dependencies with the GAN in order to assure functional placement of the key and the exit door. Their work addresses a problem GANs have with learning level semantics, but the levels are restricted in scale based on the size of training instances.

[1]J. Gutierrez is an undergraduate at Southwestern University, Georgetown, TX 78626, USA `gutierr8@southwestern.edu`

[2]J. Schrum is an Assistant Professor of Computer Science at Southwestern University, Georgetown, TX 78626, USA `schrum2@southwestern.edu`

978-1-7281-6929-3/20/$31.00 ©2020 IEEE

This paper explores a new hybrid PCG approach for dungeon crawlers based on levels from the actual *Legend of Zelda* (1986). Specifically, a GAN generates rooms based on the Video Game Level Corpus (VGLC [8]) description of the game. To scale up to large dungeons with interesting challenges, rooms are organized into a dungeon using a generative graph grammar [9] which maps a high-level, human-designed *mission* to a sequence of room obstacles, and ultimately a complete dungeon. Combining the techniques creates new and interesting dungeons of arbitrary size.

This new technique (`Graph+GAN`) was evaluated by 30 human subjects. Each played three types of dungeons to compare the enjoyability, complexity, novelty, organization, and challenges of each through surveys. They played a dungeon from the original *Legend of Zelda*, a graph grammar dungeon with rooms from the original game, and a dungeon generated with the new `Graph+GAN` technique. Players rated dungeons roughly the same in most metrics. The exception is that GAN rooms were significantly less organized, and were considered most complex by a significant number of participants.

These findings show that this technique can generate levels similar to hand-crafted dungeons from *The Legend of Zelda*. However, these dungeons also contain unique new content, and a multitude of such dungeons can be generated.

II. Related Work

Procedural dungeon generation has been a topic of interest since *Rogue* was released in 1980. As more complex games were released, the idea of procedurally generating dungeons became more prevalent. The popular games in the *Diablo* series use PCG for generating dungeons, quests, and events. These features add variety and make these games more interesting and unpredictable, increasing replayability.

Procedural generation of dungeons has been widely studied in academia [10]. Some representative techniques include cellular automata [11], various evolutionary approaches [12], [13], and generative grammars [9], [14].

Dormans used a generative graph grammar to procedurally generate a dungeon mission, and a shape grammar to generate the dungeon itself [9]. Graph and shape grammars were further explored to generate dungeons similar to *The Legend of Zelda: A Link to the Past* (LttP) in an undergraduate thesis [15]. These dungeons required particular graph and shape grammars to produce results similar to LttP. Although new dungeon layouts were created, the rooms came from LttP rather than being generated from scratch.

A recent development is the use of GANs to model the latent design space of a level corpus. Volz et al. used a

GAN to generate *Super Mario* levels with objective-based evolution [3]. A similar approach was later applied to *Doom* levels [4]. A GAN can even be replaced with an autoencoder, as was done to evolve levels for *Lode Runner* [16]. The approach worked in Mario despite a small data set, and the *Doom* and *Lode Runner* data sets were quite large.

However, for certain games it is hard to produce playable levels because of limited training data. This challenge was overcome by Park et al. with multiple GANs [5]: one GAN to create levels for a puzzle game from a small training set, and a second GAN using an augmented data set consisting of the original set plus levels from the first GAN that were actually solvable. Torrado et al. [7] used a similar approach, incorporating playable levels back into the training set when designing levels for the GVG-AI [6] version of Zelda.

In this paper, rather than make the GAN do more work, a division of labor is imposed. The GAN models the interior of individual rooms, and a generative graph grammar determines the dungeon layout and what items/obstacles are placed in each room. The result is a method that creates dungeons based off of *The Legend of Zelda*, described next.

III. The Legend of Zelda

The Legend of Zelda involves 18 dungeons across two quests (9 each) accessible via an overworld map. Each dungeon is composed of several rooms filled with enemies, items, and secret passages, where the end goal of each dungeon is find a *Triforce*, which completes the dungeon.

Each room is the same size. Although room layouts vary, many are reused both within and across dungeons. Rooms can be connected in a variety of ways: simple doors, doors requiring a key (*Lock*), doors that only open when all enemies in the room are defeated (*Soft Lock*), doors that open when a puzzle is solved (*Puzzle*), and passages that need to be bombed to open. These connections are always in a side wall of the room, though some dungeons have stairs to standalone rooms that are not part of the main map layout. Stairs are excluded from dungeons in this study.

Many interesting items can be collected in the game, but only a few are relevant to this paper: keys, hearts, bombs, and the raft. Hearts replenish a player's health. Bombs allow the player to blow up walls to reveal hidden doors or kill enemies. The raft item allows players to move across one water tile. It is introduced in Dungeon 4-1 (4th dungeon of Quest 1, Fig. 1) and used throughout the rest of the game.

Data about Zelda levels was obtained from the Video Game Level Corpus (VGLC [8]). This data provides text representations of the tiles present in each dungeon. Details of this representation, and how it maps to the one used in this paper, are in Table I. There are many symbols from the VGLC data, but since many of these tiles serve the same purpose as others, the tile training set is simplified.

IV. Dungeon Generation

A GAN is trained to generate individual rooms, which can then be combined into dungeons using a generative graph grammar. The 2D layout of the rooms is derived in part

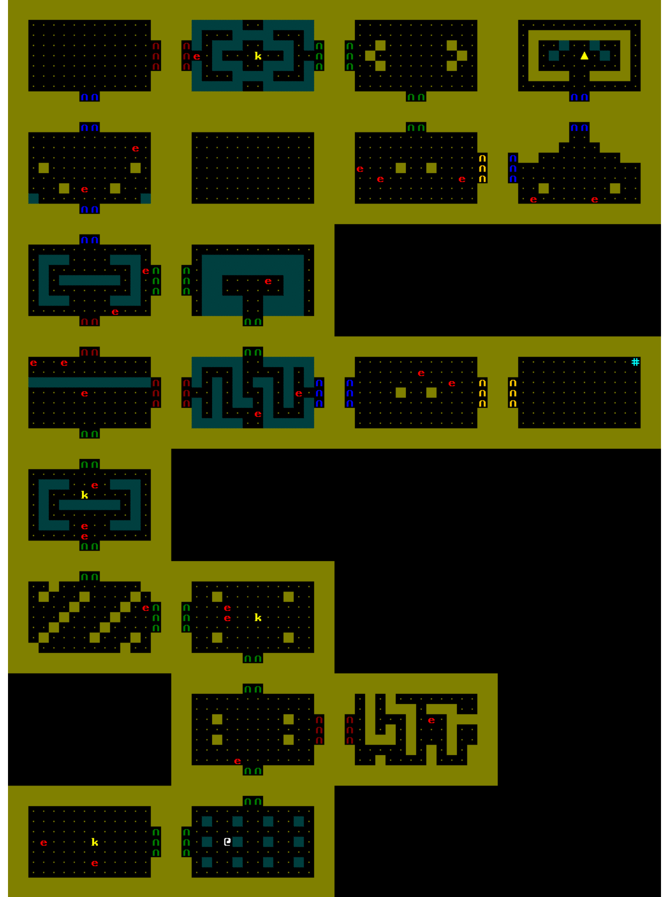


Fig. 1: Dungeon 4-1 from *Legend of Zelda* converted to the Rogue-like engine. The goal is to reach the *Triforce* (triangle) in the top-right room. In the middle right, there is a blue # item, which is the raft; it allows players to cross one water tile (dark blue), which is necessary to traverse the room three spaces to the left of the room with the raft. In the original game, the raft room was underground (via stairs), and did not appear on the map. Due to limitations of the game engine for this study, the room was directly added to the map.

from the graph. To assure that the dungeon is beatable, some additional walls may need to be knocked down. Users can then play a Rogue-like game in the repaired dungeon.

A. Zelda GAN

To generate Zelda rooms, the same GAN architecture/code used in Mario [3] is used (Fig. 2). The only differences are a change in output size to accommodate a different tile type count, and a reduced latent vector size of 10 because initial experiments indicated that an unnecessarily large latent vector led to large areas in the latent space with little variation. The output width and height were maintained at 32×32 for backwards compatibility. Zelda rooms are only 16×11, but the GAN makes the surrounding space floor tiles.

This GAN can be trained on any 2D tile-based level representation. The generator takes latent vectors of noise from $[-1, 1]^{10}$ as input, and outputs a 3D volume of 32×32 vectors of length 6. Each value in each vector corresponds to a tile type in Table I, and these vectors are collapsed so that

TABLE I: Tile Types Used in Generated Zelda Rooms.

Tile types come from VGLC, but many were unnecessary in the simplified Rogue-like engine used to play the levels. Thus the available tile set was reduced to three relevant types: floor, wall, and water. VGLC rooms were converted to use only these three tile types when serving as training input to the discriminator, and GAN outputs were used to make rooms using only these three tiles.

Tile type	VGLC	Game	Rogue-like	Rogue Type
Floor	F			Floor
Wall	W			Wall
Block	B			Wall
Door	D			Wall
Stair	S			Wall
Water	P			Water
Walk-able Water	O			Water
Water Block	I			Water
Monster Statue	M			Water

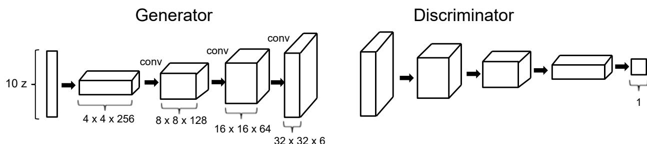


Fig. 2: The Zelda GAN architecture.

the tile at its position in the resulting 2D image corresponds to the maximum value in the vector[1]. The upper-left 16×11 portion of the image can then be interpreted as a Zelda room.

An additional discriminator network is also used during training. Its input is a one-hot encoded version of either a Zelda room from the training set, or fake output produced by the generator. Over the course of 10,000 epochs it is trained to make its single output 1 for real Zelda rooms and -1 for generated rooms. The generator itself is trained along with the discriminator, to the point where it produces convincing fake Zelda rooms. After training, the discriminator performs no better than a coin toss, and is thus discarded.

To generate the training set, the 18 dungeons in VGLC were split into rooms and encoded as GAN inputs. Because there are many repeated rooms throughout the dungeons, duplicates were eliminated. Some Zelda tiles have a similar function, but a different aesthetic. Such tiles were merged into one, as seen in Table I. The VGLC data incorrectly designates statue tiles as monsters, but the GAN interprets them as water tiles. Additionally, doors were removed from the training data, because doors need to be placed in accordance with the game mission defined by the graph grammar.

B. Graph Grammar

A generative graph grammar [9] determines how rooms connect in a dungeon. A designer-provided *backbone* graph represents the *mission* of a dungeon. The backbone includes specific rooms that must be present in the dungeon. The backbone used in this paper is *Start* $\rightarrow$ *Enemy* $\rightarrow$ *Key* $\rightarrow$ *Lock* $\rightarrow$ *Enemy* $\rightarrow$ *Key* $\rightarrow$ *Puzzle* $\rightarrow$ *Lock* $\rightarrow$ *Enemy* $\rightarrow$ *Triforce*. The backbone is a sequence of non-terminal symbols that get replaced until only terminals remain. For each pair or single symbol there is a finite set of rules defining what could replace it. For example, (*Key* $\rightarrow$ *Lock*) could be replaced with a key room that has two neighbors: a dead-end enemy room, and a locked room leading onward. Full rule set is in online material[2] (Fig. 8). Each rule defines a mini-graph that is placed into the backbone and can be made up of both non-terminals and terminals. An example of the iterative replacement process is in Fig. 3. This process can generate multiple graphs representing different dungeons, but ensures that the general sequence stays the same.

[1]Only three of six values are used. The GAN originally supported six tile types, but this setting was not changed after settling on three tile types.

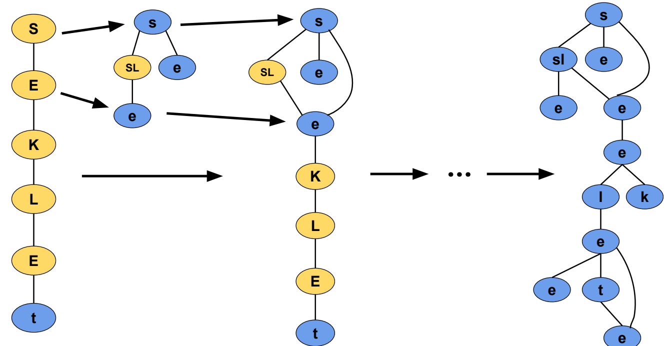


Fig. 3: Graph Expansion Example. The first two nodes of the initial graph are replaced with a randomly chosen sub-graph defined by the available graph grammar rules. Non-terminal symbols are represented as capital letters (yellow) and terminals as lower case letters (blue). The process repeats until there are no non-terminals. Nodes correspond to rooms.

TABLE II: Non-terminal Graph Grammar Symbols.

Each non-terminal symbol defines a type of room that must be in the dungeon, but during the generation process, edges connecting to non-terminal symbols get transformed to more elaborate sub-graphs that contain terminal representations of indicated rooms.

Symbol	Short	Description
Start	*S*	Dungeon starting room. Only one.
Enemy	*E*	Room with random number of enemies.
Key	*K*	Has enemies, and key appears after defeating them.
Lock	*L*	Has door that is unlocked by a key.
Soft Lock	*SL*	Has enemies, and a door that opens when they are defeated. Also contains raft item.
Puzzle	*P*	Has door that opens when puzzle block is pushed.
Triforce	*T*	Has Triforce. Dungeon complete once found. Only one.

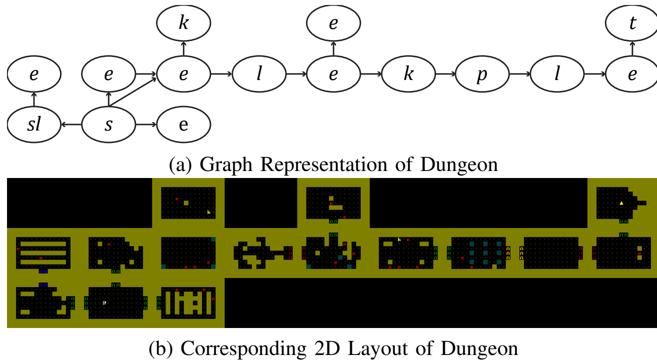
Non-terminals used by the grammar are in Table II. Not all symbols in the table are in the initial backbone, but can be added by grammar rules. The available rules assure that at least one *Soft Lock* room is in every dungeon, despite its absence from the backbone. Once a graph is created, the actual 2D layout of rooms must be determined.

C. Dungeon Layout

Dungeon rooms are placed in breadth-first order beginning with the start room of the graph. However, there may not be space around a room to accommodate its neighbors. To ensure that all rooms are placed, the algorithm backtracks if no space is available around a room needing a neighbor.

Specifically, a list of *edges* (between rooms) is generated in breadth-first order from the start room. This list is iterated

[2]southwestern.edu/~schrum2/zeldagan.html



(a) Graph Representation of Dungeon

(b) Corresponding 2D Layout of Dungeon

Fig. 4: Creation of Dungeon From Graph. (a) Graph that represents a dungeon. Each node represents a room, and each edge represents a doorway between rooms. Symbols in each node indicate the type of obstacles present in the room. The graph is directed, but the player can go back and forth between rooms. The directed edges show how the player would encounter each room for the first time. (b) The complete generated dungeon based on the above graph, with specific room layouts determined by the GAN. Note that one edge of the graph (diagonal from s to e) is lost when generating the dungeon.

through, and any room in an edge not yet placed is added to the dungeon. After the start room, all rooms must be placed in relation to the first room in an edge. A random position orthogonally adjacent to the previously placed room is chosen for its neighbor, backtracking whenever all surrounding positions of a room are occupied. Backtracking undoes the last placement and attempts an alternative that has not yet been tried. Search continues until the list of edges is exhausted. Note that only the first occurrence of each room is placed. Although the graph represents the connectivity of rooms, the 2D layout typically loses edges present in the original graph. The layout attempts to match the graph as closely as possible (Fig. 4).

Certain rooms are manipulated according to the grammar symbols. *Enemy* rooms randomly get 1–3 enemies placed in random locations. *Key* rooms have a key placed in a random empty spot in the room followed by randomly placed enemies. *Lock* rooms have a locked door placed at the connection leading to the next room. *Soft Lock* rooms have a soft locked door and randomly placed enemies. Additionally, the first soft locked room of the dungeon contains a raft item. *Puzzle* rooms have a door that can only be opened by finding and pushing a particular block in the room in a specific direction. A random spot in the room currently with or without a block becomes the puzzle block. *Triforce* room has a Triforce, represented as a yellow triangle, in the middle of the room. Bomb-able doors have a 40% chance of replacing a normal door; normal meaning that it is not locked, soft locked, or puzzle locked.

D. Room Repair

To assure that each dungeon is beatable, some rooms are modified to create a path between certain points of interest. A* search is used to check that dungeons are beatable. The A* state representation tracks puzzle blocks, keys, and the raft item, but ignores enemies and always assumes there are sufficient bombs for bomb-able walls.

If A* fails to beat a dungeon, then one room is modified. Each room has points of interest (POIs): doors, keys, puzzle blocks, the raft, and the Triforce. A* tracks the visited and unvisited POIs. On search failure, a random unvisited POI is chosen along with a random visited POI in the same room (if there were no visited POIs, then two unvisited POIs are chosen). A modified Bresenham's line algorithm [17] draws floor tiles from the visited POI to the unvisited POI. Puzzle blocks are a special case requiring POIs for both before and after the push. Afterward, A* resumes where it left off. This process repeats until A* beats the dungeon.

The repair process assures that all dungeons are playable, though only 10 of 30 GAN dungeons and 16 of 30 pure graph grammar dungeons needed any repair. Per dungeon, the average number of rooms repaired was less than one for the 30 GAN dungeons and the 30 graph grammar dungeons.

E. Rogue-like Game

To interact with the dungeons, a Rogue-like game was created in Java using the AsciiPanel library by Trystans[3]. The Rogue-like game emulates the gameplay in *Legend of Zelda*. However, the game is turn-based and only features one enemy type. Many fancy items in Zelda are absent, but there are still bombs, and every level has a raft.

All actions are turn-based, so combat is simple. The player moves first and then the enemies. If an enemy is adjacent to the player, including diagonal to it, it will attack. Each enemy attack has a 50% chance of hitting and subtracting a heart from the player. The player can only attack enemies in orthogonally adjacent positions, by pressing the appropriate arrow key. When an enemy blocks the avatar's movement, an arrow press is an attack instead of a move. When enemies are not adjacent to the player, they move toward it, but only if the player is within line of sight of 4 tiles. Otherwise, they move randomly. Enemies also move over water tiles.

Upon death, enemies sometimes drop a heart or a bomb. If the player with no bombs enters an empty room, enemies sometimes spawn so the player will be able to pick up bombs. There is at least one bomb-able wall in each dungeon.

V. Human Subject Study

The method of dungeon generation described thus far (`Graph+GAN`) is evaluated by having humans compare it to two other types of dungeon: a graph grammar dungeon that does not use a GAN (`Graph`), and Dungeon 4-1 from *Legend of Zelda* (`Original`). Whenever a `Graph` dungeon places a room, it is chosen randomly from the set of all rooms in the VGLC training set. `Graph` and `Graph+GAN` dungeons seen by each participant were different. The `Original` dungeon played by every participant was Dungeon 4-1, because it is sufficiently interesting to represent a meaningful comparison. Some earlier dungeons are simplistic in comparison, and

[3]`http://trystans.blogspot.com/`

many later dungeons are so large that having users play them would be too time consuming. Dungeon 4-1 is also ideal because its raft item allows players to traverse obstacles in a new way, whereas many of the special items in other dungeons are weapons that introduce combat mechanics difficult to emulate in the Rogue-like engine.

The study had 30 participants (university students, faculty, and staff). Each participant played through a dungeon of each of the three types in a different order (5 per each of 6 possible orders). After each dungeon, the participant took a survey ranking the dungeon on a 1–5 scale in various categories. After the second dungeon, users indicated which of the two were better in various respects, and after the third dungeon all three were ranked relative to each other. Participants also provided open-ended text responses at each stage.

Players start each dungeon with 0 bombs, 0 keys, and 4 hearts. It was possible to die, in which case the user would start the dungeon over, but the game would be easier. The starting/max number of hearts would increase, as would the chance of defeated enemies dropping a heart pickup. After dying, the starting hearts would increase to 6, then 8, then 20. Unexpectedly, one participant did not finish one of the dungeons even with this many tries, and thus repeated the dungeon starting over at 4 hearts. The heart drop rate for defeated enemies started at 30%, and increased with each death to 60%, then 90% for the remaining deaths.

Source code for running the user trials as well as video of the trials is accessible here: `southwestern.edu/~schrum2/zeldagan.html`.

VI. Results

Statistical analysis of numerical ratings and relative rankings is provided, as are objective measures of the novelty of rooms in each dungeon type. Qualitative user responses provide additional insight into the quantitative data.

A. Numeric Participant Ratings

`Graph` and `Graph+GAN` dungeons are comparable to `Original` in most respects. Kruskal-Wallis tests ($df = 2$) indicate that there are no significant differences between dungeon types in terms of enjoyability ($H = 1.5065, p = 0.4708$), challenge in finding the exit ($H = 2.5478, p = 0.2797$), challenge from enemies ($H = 1.2331, p = 0.5398$), map complexity ($H = 2.8105, p = 0.2453$), room complexity ($H = 1.2279, p = 0.5412$), and room novelty ($H = 4.2023, p = 0.1223$). Only in terms of room organization is there a significant difference between dungeon types ($H = 11.337, p = 0.003454$), and post-hoc pairwise Mann-Whitney U tests with FDR error correction show that it is specifically the `Graph+GAN` rooms that are less organized than rooms of both `Original` ($p = 0.0056$) and `Graph` ($p = 0.0164$). Since `Original` and `Graph` make use of the same set of rooms, there is no significant difference in their level of organization ($p = 0.4866$). Distributions of participant ratings for each dungeon type in all categories are shown as violin plots in Fig. 5.

B. Relative Participant Rankings of Dungeons

After all three dungeons, participants ranked dungeons in terms of enjoyment, room complexity, room novelty, map layout challenge level, and chaos of the rooms (Fig. 6). For each category the number of *Most* and *Least* ratings for each dungeon type were compared using exact multinomial tests.

There is no significant difference in *Most* ratings in the categories of enjoyment ($p = 0.185$), room novelty ($p = 0.2622$), map challenge ($p = 0.7647$), or room chaos ($p = 0.07238$). The null hypothesis that was *not* rejected is that the 30 user ratings are evenly split into 10 per dungeon type. Only for room complexity was there a significant difference between *Most* ranks ($p = 0.005532$). Post-hoc pairwise binomial tests with FDR error correction indicate that `Graph+GAN` rooms received significantly more *Most Complex* ranks than `Graph` (17 vs. 3, $p = 0.0077$), though the ratings in Fig. 5 indicate that the degree of the difference is small. Also, despite `Graph` and `Original` rooms coming from the same set, there is no significant difference between the number of *Most Complex* ranks of `Original` vs. `Graph+GAN` (10 vs. 17, $p = 0.2478$). The difference between `Original` and `Graph` was also not significant (10 vs. 3, $p = 0.1384$).

For *Least* ranks, there was no significant difference in enjoyment ($p = 0.1117$), room complexity ($p = 0.156$), room novelty ($p = 0.5943$), or room chaos ($p = 0.0724$). However, there was a significant difference in map challenge ($p = 0.0184$). Specifically, post-hoc binomial tests with FDR correction indicate that `Graph` received significantly more *Least Challenging* ranks than `Original` (17 vs. 4, $p = 0.022$). This finding is interesting because the layouts for `Graph+GAN` and `Graph` were defined by the same algorithm. However, the differences between `Original` and `Graph+GAN` (4 vs. 9, $p = 0.267$) and `Graph` and `Graph+GAN` (17 vs. 9, $p = 0.253$) were not significant.

Despite few distinctions being statistically significant, there are interesting non-significant differences. First, 15 users found `Original` most enjoyable. However, when compared with the 1–5 ratings in Fig. 5, it seems that the degree to which `Original` was more enjoyable was minor. In contrast, 16 participants found `Graph+GAN` rooms most chaotic. The 1–5 ratings for *Room Organization* relate to these responses, and indicate that GAN rooms may actually be moderately more chaotic/less organized. `Original` received the highest number of *Most Novel* ranks (13) and smallest number of *Least Novel* ranks (8) with respect to its rooms, whereas `Graph` received the most *Least Novel* ranks (13) and least *Most Novel* ranks (6). This contrast is strange because every room in `Original` is a room that could be in `Graph` dungeons. The GAN generated rooms, often unique to these dungeons, were ranked *Most Novel* 11 times and *Least Novel* 9 times. This confusion can be clarified with the objective measure of novelty presented next.

C. Objective Novelty Comparisons

An objective calculation of room novelty was made to measure differences between dungeon types. *Room Novelty*

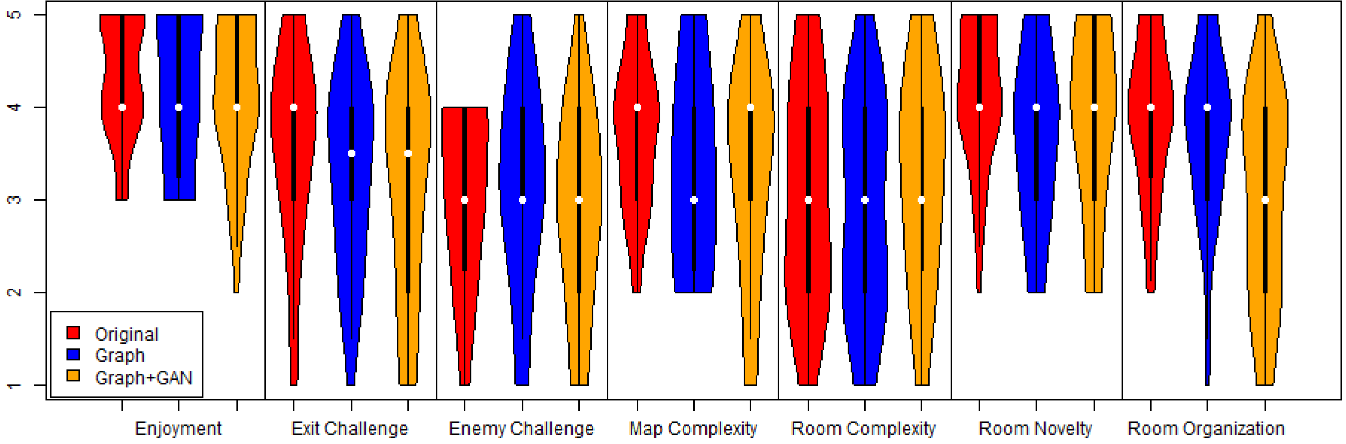


Fig. 5: Participant Ratings Of Each Dungeon Type. Violin plots depict distributions of participant ratings on a 1–5 scale for properties of each dungeon type. Each group of plots shows ratings of `Original`, `Graph`, and `Graph+GAN`. The aspects being rated are under each group. The only category with a statistically significant difference is *Room Organization*: `Graph+GAN` rooms are less organized than others.

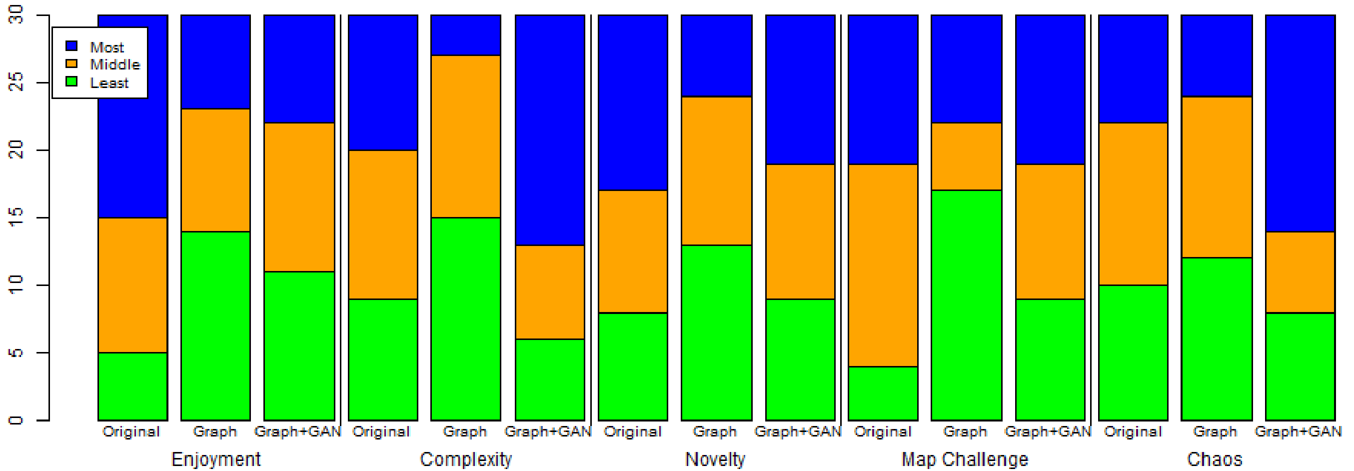


Fig. 6: Participant Relative Rankings Of Each Dungeon Type. Stacked bar charts show the number of participants that assigned each dungeon type a particular rank with respect to each other in each category. Categories are listed along the bottom. Each bar shows the count that ranked the given dungeon type as *Least*, *Middle*, and *Most* from bottom to top. Some notable observations are that 15 participants rated `Original` as most enjoyable, 17 rated `Graph+GAN` as most complex, and 16 rated `Graph+GAN` as most chaotic. In contrast, 14 rated `Graph` as least enjoyable, 15 rated `Graph` as least complex, and 17 rated `Graph` map layouts as least challenging.

is the average normalized distance of that room from all other rooms in its dungeon. The distance metric is the count of tile positions in which two rooms differ. Only the novelty of the primary 12×7 floor area is considered (excluding walls and doors). *Dungeon Novelty* is the average novelty of all rooms in the dungeon. Summary novelty statistics are in Table III.

Comparing novelty scores of different dungeon types using one-way ANOVA reveals significant differences ($F(2, 75) = 7.317, p = 0.00125$). Post-hoc pairwise comparisons with Tukey's HSD and error-adjusted p-values are presented. Even though `Graph` only uses rooms from the original game, `Graph` is significantly more novel than `Original` ($p = 0.00854$). Here, `Original` refers to all 18 dungeons from the original game. `Graph+GAN` is also significantly more novel than `Original` ($p = 0.00116$), but not significantly different from `Graph` ($p = 0.7377$).

The novelty of Dungeon 4-1 specifically is 0.2970, which is higher than the averages for all dungeon types; a very novel dungeon was used in this study. Users explicitly mentioned this: "I enjoyed that the layout was different in almost all rooms." Fig. 1 verifies this, and indicates why users rated the novelty of this dungeon high, even though the set of all rooms in the original game has low novelty.

In addition to calculating novelty scores for each dungeon, averages across all rooms present in a given collection of dungeons can also be calculated (Table III). ANOVA indicates a significant difference between the room novelties of all rooms from the original game, all rooms in all 30 `Graph` dungeons, and all rooms in all 30 `Graph+GAN` dungeons ($F(2, 1439) = 47.85, p < 0.0001$). Tukey's HSD once again indicates that `Graph` and `Graph+GAN` are significantly more novel than `Original` ($p < 0.0001$), but

TABLE III: Objective Novelty Scores.

Summary statistics of novelty scores for different collections of dungeons and rooms are shown. N is the sample size. The first three rows are based on *Dungeon Novelty*, and the next six on *Room Novelty*. Calculations are performed across all rooms in the given collections, and across only the unique rooms. `Original` is less novel, unless you focus on unique rooms only.

Type	N	Avg ± StDev	Min	Max
`Original` Dungeons	18	0.2348 ± 0.0496	0.1311	0.3178
`Graph` Dungeons	30	0.2752 ± 0.0485	0.1975	0.3759
`Graph+GAN` Dungeons	30	0.2837 ± 0.0357	0.1970	0.3899
All `Original` Rooms	459	0.2481 ± 0.1118	0.1545	0.6733
All `Graph` Rooms	491	0.2941 ± 0.0772	0.2035	0.5920
All `Graph+GAN` Rooms	492	0.3019 ± 0.0816	0.2108	0.5891
Unique `Original` Rooms	38	0.3442 ± 0.0888	0.2453	0.6062
Unique `Graph` Rooms	87	0.3437 ± 0.0713	0.2471	0.5334
Unique `Graph+GAN` Rooms	367	0.3268 ± 0.0720	0.2337	0.5802

not significantly different from each other ($p = 0.3751$).

Calculations on sets of only the unique rooms of each collection are also performed (Table III), because these collections have many repeated rooms, especially those in `Original`. Although `Graph` uses the same rooms, they are sometimes modified by the repair process (Section IV-D), so `Graph` has more unique rooms. `Graph+GAN` has the most unique rooms. When reduced to only unique rooms, there is no significant difference among types ($F(2, 489) = 2.517, p = 0.0818$), indicating that `Original` dungeons reuse certain rooms more heavily than the random sampling of `Graph` or the GAN output of `Graph+GAN`.

D. Informative Participant Quotes

Quotes contextualize the quantitative findings. In particular, why was Dungeon 4-1 appealing? Participants enjoyed the water obstacle that was only passable with the raft. One said, "water cross tool/item was enjoyable." Another said, "I liked that you had to wait later in the level to get the water walking thing and that helped you get further in the level."

More generally, backtracking was appealing, as indicated about a `Graph` dungeon: "I liked the need to backtrack through a couple of the dungeon rooms for necessary items if you didn't find them first." However, for the graph backbone in this study, only some generated levels required the raft to be beaten. In others, players found the raft before needing it. One user said of a `Graph` dungeon, "I liked that this dungeon had rooms that used the raft more than the other; however, I got the raft early enough to where I didn't have to worry about water."

Better design of the graph backbone could enforce backtracking as in Dungeon 4-1. However, some users appreciated how expectations were subverted: "I liked that there was the raft item near the beginning of the dungeon that I could see but couldn't reach. I felt like I had to figure out a way to get to the raft, but couldn't." Of a `Graph+GAN` dungeon: "This dungeon was very chaotic, with items you didn't need in places you couldn't access. I liked that a lot because it threw me off and had me thinking about different possibilities."

This quote supports data indicating that GAN rooms are less organized. A participant observed: "there were parts of rooms and enemies that I couldn't reach." This oddity could be avoided by restricting item and enemy placement to reachable locations. Reachability aside, many GAN rooms simply look more chaotic: "There was a large mix of wall and water blocks, in ways that didn't seem completely natural. There was very little symmetry and a lot of obstacles."

Although the GAN produces chaotic rooms, 10 participants specifically said things like "They seemed organized," and "I felt like the rooms were organized." The GAN also produces rooms from the original training set, and unique rooms that have a level of structure similar to original rooms (Fig. 7). Randomness led to some users seeing more rooms of one type than the other. Some people appreciated the chaotic rooms: "they were chaotic but in a good way, none seemed like a copy of the previous and kept me on my toes."

Much criticism was directed at `Graph` dungeon layouts. Participants said, "I didn't enjoy how simple the dungeon was overall," "The map layout was very simple, not very novel," and "this one favored simpler layouts." `Graph+GAN` dungeons did not receive many comments like this, despite using the same graph grammar. Randomness in generation may have played a role, though it may be that chaotic GAN room layouts distracted from issues with the dungeon layout.

The most criticized layout was a linear layout without much branching: "just a diagonal line, not many choices," and "It was not as difficult to make it from room to room due to the lack of multiple bordering squares." These complaints could be remedied by having segments for the dungeon backbone with more diverse path options. However, the main issue seems to be randomness in the 2D layout, because some `Graph` dungeons *had* interesting layouts: "The map layout had me thinking of different areas the secret doors were in. It was interesting to try and figure out where to go next."

Ultimately, conflicting opinions about several aspects of `Graph` and `Graph+GAN` dungeons are likely based partly on differing user preferences and perspectives, but are potentially also based on the variety of dungeons that can be produced by these methods, making it hard to categorize all dungeons of either type in the same way.

VII. Discussion and Future Work

The `Graph+GAN` technique presented in this paper procedurally generates dungeons similar in terms of enjoyment, challenge level, and complexity to Dungeon 4-1 from *The Legend of Zelda*. Dungeon 4-1 is special because it introduces the raft item which makes new types of puzzles possible. Creating dungeons comparable to this dungeon is impressive. Furthermore, the `Graph+GAN` technique can create an effectively infinite multitude of such dungeons.

Improving the handcrafted backbone for the graph grammar could vastly improve layouts, remedying many user complaints. The dungeon generation method would be the same, but a better designer could encourage the method to produce better output. Tweaking the backbone requires relatively little effort, given that the benefit is an infinite multitude of levels. The backbone could be adjusted to force backtracking after obtaining the raft, and could provide any desired number of locked doors and/or puzzle rooms. Without a graph grammar, a designer can fix a specific level,

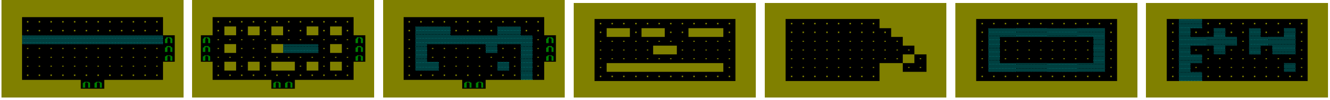


Fig. 7: Spectrum of Rooms Generated by the GAN. Some are identical or nearly identical to rooms in the training set, but others seem less structured and predictable, thus showcasing the diversity of the GAN outputs, but also revealing why its rooms are sometimes considered chaotic and unorganized.

but needs to expend great effort to create whole new levels adhering to a particular high-level design plan.

Both `Graph` and `Graph+GAN` techniques produce a multitude of levels, but `Graph` makes repetitive use of the same rooms. Even when it produces a layout as interesting as an `Original` level, it offers nothing new in terms of rooms. In contrast, GAN rooms are less organized, and considered most complex. Some users enjoyed the unpredictability of certain GAN rooms, but the GAN can also produce structured rooms similar to those from the original game.

In the future, it is desirable to have a data-driven method replace the graph grammar entirely. Whether GANs or some other method can be adapted for this purpose is uncertain. The variation across the 38 unique rooms in the original game (training set) seems less than the variation across the 18 dungeons. There is a combinatorial explosion of potential complexity in complete dungeons when the variety of possible rooms is taken into account, and 18 dungeons of very different sizes may not be enough for a GAN to learn general design principles. However, bootstrapping methods (that work with limited data) for applying GANs to level design are an area of active research [7], [5]. Generating the entire dungeon based on data will hopefully better capture design patterns of the original dungeons.

Rather than having a simple GAN generating walls and water blocks, it would be desirable to have generated enemies, keys, puzzle blocks, etc. For further research, a conditional GAN [18] could be used to generate rooms based on type. For instance, enemy rooms or puzzle rooms could be specifically requested. This approach would better match the original game's enemy, puzzle block, and key placement.

Though this paper shows the potential of the `Graph+GAN` approach, a more impressive example would utilize all details of Zelda's levels, and create a gameplay experience closer to the original. Unfortunately, the VGLC data is lacking many details. However, the current GAN model could, without modification, generate rooms for a more intricate game if the gameplay engine were more complex. Tweaks to the engine could also enable online play, making the game more accessible for future studies.

VIII. Conclusions

A new hybrid approach to generating game dungeons combining a Generative Adversarial Network with a Generative Graph Grammar was presented and validated with a user study. User responses indicate that results were comparable to a handcrafted level from *The Legend of Zelda*. Better design of the graph backbone, and a more sophisticated game engine could result in a more impressive experience. This new approach to Procedural Content Generation could prove valuable for commercial video games.

Acknowledgments

This research is supported in part by the Summer Collaborative Opportunities and Experiences (SCOPE) program, funded by various donors to Southwestern University.

References

[1] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games*, 1st ed. Springer, 2016.
[2] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Nets," in *Neural Information Processing Systems*, 2014, pp. 2672–2680.
[3] V. Volz, J. Schrum, J. Liu, S. M. Lucas, A. M. Smith, and S. Risi, "Evolving Mario Levels in the Latent Space of a Deep Convolutional Generative Adversarial Network," in *Genetic and Evolutionary Computation Conference*. ACM, 2018.
[4] E. Giacomello, P. L. Lanzi, and D. Loiacono, "Searching the Latent Space of a Generative Adversarial Network to Generate DOOM Levels," in *Conference on Games*. IEEE, 2019.
[5] K. Park, B. W. Mott, W. Min, K. E. Boyer, E. N. Wiebe, and J. C. Lester, "Generating Educational Game Levels with Multistep Deep Convolutional Generative Adversarial Networks," in *Conference on Games*. IEEE, 2019.
[6] D. Perez-Liebana, J. Liu, A. Khalifa, R. D. Gaina, J. Togelius, and S. M. Lucas, "General Video Game AI: A Multitrack Framework for Evaluating Agents, Games, and Content Generation Algorithms," *Transactions on Games*, vol. 11, no. 3, pp. 195–214, Sep. 2019.
[7] R. R. Torrado, A. Khalifa, M. C. Green, N. Justesen, S. Risi, and J. Togelius, "Bootstrapping Conditional GANs for Video Game Level Generation," *arXiv*, vol. abs/1910.01603, 2019.
[8] A. J. Summerville, S. Snodgrass, M. Mateas, and S. Ontañón, "The VGLC: The Video Game Level Corpus," in *Procedural Content Generation in Games*. ACM, 2016.
[9] J. Dormans, "Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games," in *Procedural Content Generation in Games*. ACM, 2010.
[10] R. van der Linden, R. Lopes, and R. Bidarra, "Procedural Generation of Dungeons," *Transactions on Computational Intelligence and AI in Games*, vol. 6, no. 1, pp. 78–89, March 2014.
[11] L. Johnson, G. N. Yannakakis, and J. Togelius, "Cellular Automata for Real-Time Generation of Infinite Cave Levels," in *Procedural Content Generation in Games*. ACM, 2010.
[12] V. Valtchanov and J. A. Brown, "Evolving Dungeon Crawler Levels with Relative Placement," in *C* Conference on Computer Science and Software Engineering*. ACM, 2012, pp. 27–35.
[13] D. Ashlock, C. Lee, and C. McGuinness, "Search-Based Procedural Generation of Maze-Like Levels," *Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 260–273, 2011.
[14] J. Dormans and S. Bakkes, "Generating Missions and Spaces for Adaptable Play Experiences," *Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 216–228, 2011.
[15] R. Lavender, "The Zelda Dungeon Generator: Adopting Generative Grammars to Create Levels for Action-Adventure Games," University of Derby, Tech. Rep., 2015, Undergraduate Thesis.
[16] S. Thakkar, C. Cao, L. Wang, T. J. Choi, and J. Togelius, "Autoencoder and Evolutionary Algorithm for Level Generation in Lode Runner," in *Conference on Games*. IEEE, 2019, pp. 1–4.
[17] J. E. Bresenham, "Algorithm for Computer Control of a Digital Plotter," *IBM Systems Journal*, vol. 4, no. 1, pp. 25–30, Mar. 1965.
[18] M. Mirza and S. Osindero, "Conditional Generative Adversarial Nets," *arXiv*, vol. abs/1411.1784, 2014.