Using Multiple Generative Adversarial Networks to Build Better-Connected Levels for Mega Man

Benjamin Capps
Southwestern University
Georgetown, Texas, USA
cappsb@southwestern.edu

Jacob Schrum
Southwestern University
Georgetown, Texas, USA
schrum2@southwestern.edu

ABSTRACT

Generative Adversarial Networks (GANs) can generate levels for a variety of games. This paper focuses on combining GAN-generated segments in a snaking pattern to create levels for Mega Man. Adjacent segments in such levels can be orthogonally adjacent in any direction, meaning that an otherwise fine segment might impose a barrier between its neighbor depending on what sorts of segments in the training set are being most closely emulated: horizontal, vertical, or corner segments. To pick appropriate segments, multiple GANs were trained on different types of segments to ensure better flow between segments. Flow was further improved by evolving the latent vectors for the segments being joined in the level to maximize the length of the level's solution path. Using multiple GANs to represent different types of segments results in significantly longer solution paths than using one GAN for all segment types, and a human subject study verifies that these levels are more fun and have more human-like design than levels produced by one GAN.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; **Generative and developmental approaches**; **Learning latent representations**; **Genetic algorithms**.

KEYWORDS

Mega Man, Generative Adversarial Networks, Procedural Content Generation via Machine Learning, Neural Networks

1 INTRODUCTION

Generative Adversarial Networks (GANs [8]) are capable of reproducing certain aspects of a given training set. GANs are artificial neural networks that can be trained to generate fake samples based on real examples. Past successes include the generation of fake

celebrity faces [10] and fingerprints [3]. In the domain of games, GANs have generated levels for Mario and others [7, 9, 25, 26].

In this paper, GANs are used to generate Mega Man levels based on levels in the original game. Data from the Video Game Level Corpus (VGLC [22]) is used to train GANs. Volz et al. [26] generated Mario levels by placing individual segments left-to-right. In contrast to Mario, Mega Man levels have a snaking pattern of horizontal, vertical, and corner segments. Therefore, different GANs are used for different segment types, resulting in levels with better flow and organization, and a more human-like design.

Levels were optimized using latent variable evolution (LVE [2]): one real-valued vector consisting of the concatenation of multiple latent vectors was used to generate several segments. The vector also contained information on the relative placement of each segment. Levels were evolved using multiple objectives by Non-Dominated Sorting Genetic Algorithm II (NSGA-II [5]), the most relevant being solution path length as determined by A* Search.

Our new approach, MultiGAN, trains distinct GANs on different portions of the training data, and queries the appropriate GAN for each segment type as needed. MultiGAN is compared with the standard approach of training one GAN on all data: OneGAN. Although segments produced by GANs make no assumptions regarding neighboring segments, LVE encourages sensible transitions between segments. However, MultiGAN more easily selects appropriate segments, resulting in significantly longer solution paths, and levels that flow in a more human-like fashion. In contrast, One-GAN levels are chaotic and have shorter solutions, since irregular boundaries and undesirable shortcuts emerge despite evolution.

A human subject study was also conducted that indicates Multi-GAN levels are significantly more fun and human-like in their design than OneGAN levels, confirming our analysis of the levels, and indicating that MultiGAN is a promising approach for generating better levels for classic platforming games.

2 RELATED WORK

Procedural Content Generation (PCG [24]) is an automated way of creating content for video games. PCG via Machine Learning (PCGML [12]) is a way of combining machine learning with PCG.

Generative Adversarial Networks are an increasingly popular PCGML method for video game level generation. After the GAN is properly trained, randomly sampling vectors from the induced latent space generally produces outputs that could pass as real segments of levels. However, some segments are more convincing than others, or have other desirable qualities (e.g. enemy count, solution length, tile distribution), so it makes sense to search the latent space for these desirable segments via methods such as evolution.

The first latent variable evolution (LVE) approach was introduced by Bontrager et al. [3]. In their work a GAN was trained on a set of real fingerprint images and then evolutionary search was used to find latent vectors that matched subjects in the data set. Because the GAN is trained on a specific target domain, most generated phenotypes resemble valid domain artifacts. The first LVE approach to generating video game levels was applied to Mario [26]. Work quickly followed in other domains.

Giacomello et al. [7] used a GAN to generate 3D levels for the First-Person Shooter Doom. Gutierrez and Schrum [9] used a Graph Grammar in tandem with a GAN to generate dungeons for The Legend of Zelda. Work in the GVG-AI [16] variant of Zelda was also done by Torrado et al. [25] using conditional GANs. This work used bootstrapping to help in the generation of more solvable levels. A similar bootstrapping approach was also used by Park et al. [15] in an educational puzzle-solving game. Additional work has also been done in a broader collection of GVG-AI games by Kumaran et al. [11], who used one branched generator to create levels for multiple games derived from a shared latent space.

Additional work has also been done in the original Mario domain. In particular, Fontaine et al. [6] and Schrum et al. [19] have both used the quality diversity algorithm MAP-Elites [14] to find a diversity of interesting Mario levels. The approach by Schrum et al. specifically used Compositional Pattern Producing Networks [20] with GANs to make levels with better cohesion and connectivity. This approach was applied to Zelda as well as Mario. Schrum et al. [18] also combined GANs with interactive evolution in these domains to search the space of levels according to user preferences.

Previous work with GANs in Mario [6, 18, 19, 26] all learn to generate one level segment at a time, before placing the adjacent segments left-to-right. Mega Man levels are more complicated in that they have a snaking-pattern that can move right, up, down, and even to the left. The only other paper that has addressed this challenge is recent work by Sarkar and Cooper [17] that uses Variational Autoencoders (VAEs). This method was applied to horizontal Mario levels, vertical Kid Icarus levels, and snaking Mega Man levels. Although their results seem promising, they often still have problems with barriers between segments. In their work, A* paths through the training levels were part of training data, and proposed paths are actually part of the VAE output. However, these paths are not always valid, and example levels shown in the paper are not always traversable. Therefore, in our work, levels are specifically optimized to maximize the resulting A* path length, to assure that the Mega Man levels are actually traversable.

3 MEGA MAN

Mega Man (Rockman in Japan), was released in 1987 on the Nintendo Entertainment System (NES). Gameplay involves jumping puzzles, killing enemies, and a boss at the end of each level. Mega Man's success led to numerous sequels on the NES and other systems, most with the same graphics aesthetic and game mechanics.

The Video Game Level Corpus (VGLC [22]) contains data from numerous games, including Mega Man. VGLC is the source of training data for much of the research in Section 2 [7, 9, 17–19, 26]. The levels are represented as text files, where each character represents a different tile type from the level, as seen in Table 1. The

Table 1: Tile Types Used in Mega Man.

Tile types come from VGLC, though additional types were added based on observations of the actual game. The original VGLC did not include enemies, level orbs, or water. The "In VGLC" column indicates whether the tile was originally represented in VGLC. "Training" indicates whether the tile was used in GAN training sets. "Char" is the original character code representation in VGLC (or a made up code for tiles not in VGLC), and "Int" is a numeric code used in JSON representations of the training data. In VGLC, one tile was associated with a Cannon obstacle. That tile maps to Int 6, but was deemed unnecessary and becomes a solid block in generated levels. Additionally, although no specific enemy was used for training, a single general enemy type with code 11 was used as a placeholder for an enemy, and an algorithm later specified the type based on location.

Tile type	In VGLC	Training	Char	Int	Game
Air	Yes	Yes	–	0	
Solid	Yes	Yes	#	1, 6	
Ladder	Yes	Yes			2	
Hazard	Yes	Yes	H	3	
Breakable	Yes	Yes	B	4	
Moving Platform	Yes	Yes	M	5	
Orb	No	No	Z	7	
Player	Yes	No	P	8	
Null	Yes	Yes	@	9	
Water	No	Yes	W	10	
Generic Enemy	No	Yes	Varies	11	N/A
Ground Enemy	No	No	G	11	
Wall Enemy	No	No	W	12	
Flying Enemy	No	No	F	13	

original VGLC data is slightly modified and enhanced as described in the table caption. Additionally, a level orb is added to mark the end of each level (bosses not included). The snaking pattern of some levels presents an interesting challenge for level generation.

Mega Man Maker[1] is a fan-made game for building and playing user-created levels. The game includes content from each Mega Man game, including a platform-gun that allows the player to traverse otherwise difficult jumping puzzles with more ease. Mega Man Maker is used to visualize and play levels generated by the GANs.
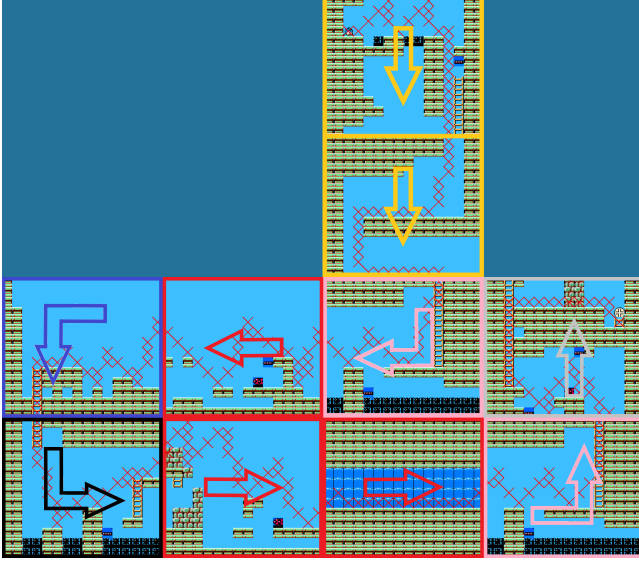
4 APPROACH

First, data was collected from VGLC for training the GANs. One-GAN was trained in a manner similar to previous PCGML work with GANs, but MultiGAN required the collected training data to be separated by type. However, levels were evolved for both approaches using the same genome encoding.

4.1 Collecting Data

MultiGAN requires the training data to be categorized by type: up, down, horizontal, lower left, lower right, upper left, and upper right segments (Fig. 1). Up and down segments are distinct because down segments often involve falling, thus making it impossible to move upward. Up segments require ladders and/or platforms that enable upward movement. In contrast, there is no distinction between left and right segments, which are both horizontal.

---

[1]https://megamanmaker.com/

Figure 1: Level Generated by MultiGAN. Each color represents a segment of a different type: Yellow:Down, Pink:Lower Right, Red:Horizontal, Blue:Upper Left, Black:Lower Left, White:Up. Upper Right segments can also be generated, but none are shown. Each GAN was trained only on the data of that given type.

The VGLC data was missing many details from the original game, some of which were added back to the data before training (Table 1). In contrast, player spawn points were replaced with air tiles. Only one is needed per level, which is better placed using simple rules.

The modified level files were scanned with a sliding window to create training data. The window size corresponds to the area visible on screen, which is 14 tiles high by 16 tiles wide. The window slides one tile at a time in the appropriate direction given where the path of the level led next. Although adjacent segments can be orthogonally adjacent in any direction, there is no branching, so there is always exactly one direction to head in toward the end of the level. Horizontal, up, and down segments are categorized according to the direction the window slides while collecting the data. A segment is identified as a corner segment whenever the direction of sliding has to change. However, each corner segment is also considered a horizontal, up, or down segment, depending on which direction the window slides when entering the segment. The result of this process is a collection of 14×16 segment training samples. OneGAN is trained with all of the data, whereas MultiGAN had a separate training set for each segment type (Table 2).
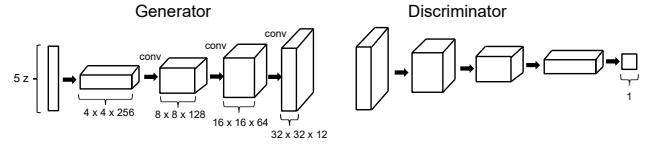
4.2 Training the GANs

The type of GAN used is a Deep Convolutional GAN, specifically a Wasserstein GAN [1], as used in previous studies [9, 19, 26]. The specific architecture is shown in Fig. 2.

There are two key components to training a GAN: a generator and a discriminator. The generator is the GAN itself, and is responsible for generating fake outputs based on an input latent vector. The discriminator is trained to recognize whether a given input is real (from the training set) or fake (from the generator).

Table 2: Characterization of VGLC Mega Man Data.

Number of segments of each type in the training data. OneGAN uses all training data, but MultiGAN uses a separate training set for each of seven distinct GANs, each consisting of samples with the same segment type.

GAN	Type	Segment Count
OneGAN	All	2344
MultiGAN	Horizontal	1462
	Up	518
	Down	364
	Upper Left Corner	10
	Lower Right Corner	9
	Upper Right Corner	8
	Lower Left Corner	8

Figure 2: GAN architecture.

The 2D JSON training segments of size[2] 14×16 are expanded into 3D volumes by converting each integer into a one-hot vector across 12 channels, one per tile type in Mega Man. During training, the discriminator is shown real and fake level segments. Fake segments are generated by the GAN by giving it randomized latent vectors of length 5. Discriminator weights are adjusted to be more accurate, and generator weights are adjusted to produce better fakes. The GAN is trained for 5000 epochs, at which point the discriminator can no longer determine whether an image is real or fake, and is thus discarded. However, the generator can now produce convincing fake level segments given arbitrary latent vector inputs.

The OneGAN is trained on all data, which is the norm, but as a result there is no clear way to retrieve a segment of a desired type. MultiGAN is trained on the same data, but each individual GAN of the MultiGAN was trained on a different category of data from Table 2. Each of the seven GANs had the same architecture (Fig. 2) and therefore same latent input size of 5 as OneGAN. Each was also trained for 5000 epochs.

4.3 Genome Encoding

Complete levels are generated from MultiGAN and OneGAN in the same way. The genome is a vector of real numbers from $[-1, 1]$ divided into sections for each segment. In each section of 9 variables, the first 5 are latent variables, and the last 4 determine the relative placement of the next segment. These 4 values correspond to up, down, left, right placement. Whichever direction has the maximum value is chosen for the next placement, unless that location is occupied, in which case the next highest value is chosen, and so on until an unoccupied neighboring location is found. If there are no unoccupied neighbors, then level generation terminates early.

For each segment generated by OneGAN, the 5 latent variables of the genome section are sent to the GAN to produce the segment. For MultiGAN, the direction of the next placement and the previous

---

[2]The actual size of the generator output and discriminator input is 32×32 for compatibility with previous research. Inputs are padded with zeros to be the right size.

placement determine the appropriate type for the current segment, and the specific GAN for the needed type is used to generate the segment. If the new direction is different from the previous one, a corner GAN is used to generate that segment. If a segment is generated to the right, and the new direction will be up, then the lower-right GAN will generate a segment to the right to prepare for the new up segment, and so on.

For simplicity, the player spawn is placed in the upper-left most section of the first segment placed, and the level ending orb is placed in the lower-right most section of the last segment placed.

5 EXPERIMENTS

This section goes into detail regarding how levels were evolved and evaluated based on desirable properties. Both the OneGAN and MultiGAN methods use NSGA-II [5] to evolve suitable levels. The two fitness objectives are solution path length and connectivity, both determined by A* search. Parameters for evolution are also explained in this section. Code for the experiments is available as part of the MM-NEAT software framework[3].

5.1 NSGA-II

NSGA-II [5] is a Pareto-based multi-objective evolutionary optimization algorithm. NSGA-II uses the concept of Pareto Dominance to sort populations into Pareto layers based on their objective scores. If an individual is at least as good as another in all objectives, and strictly better in at least one objective, then that individual *dominates* the other. The most dominant Pareto layer contains individuals which are not dominated by any other individual in the population, and is known as the nondominated Pareto front.

NSGA-II uses elitist selection favoring individuals in the Pareto front over others. The second tier of individuals consists of those that would be nondominated if the Pareto front were absent, and further layers can be defined by peeling off higher layers in this fashion. When ties need to be broken among individuals in the same layer, individuals that are maximally spread out to lower density regions of the current layer are preferred.

Pareto-based algorithms like NSGA-II are useful when objectives can be in conflict with each other, thus causing trade-offs. However, even when objectives correlate fairly well with each other, as in this paper, NSGA-II is useful in that it can provide a fitness signal in regions of the search space when one objective may be flat, without the need to define any kind of weighting between objectives.

5.2 Fitness Functions

Levels are evolved with a combination of connectivity score and solution path length: the connectivity score provides a smooth gradient to improvement, even in regions of the search space filled with unbeatable levels whose solution path length is undefined.

Solution path length is determined by A* search on a simplified model of the game to allow for quick execution. If A* could not beat the level, then it was assigned a score of -1, and levels that were beatable were assigned a score equal to the length of the A* path.

When multiple levels are unbeatable, connectivity provides a way of differentiating them. Connectivity can also differentiate two levels with the same solution length. Connectivity score is formally

---

[3]https://github.com/schrum2/MM-NEAT

defined as the proportion of traversable tiles (e.g. air, ladders) in the level that are reachable. Higher connectivity implies more places to explore, thus containing less wasted/unused space.

The simplified A* model only allows the avatar to move discretely through the space one tile position at a time, and has a simplified jumping model that simulates the playable game. The model recognizes that Mega Man will die from contact with spikes or due to falling off the edge of the screen, but ignores the existence of enemies in the level. Though the model is not perfect, it is sufficient, and faster to calculate than an actual simulation of the full dynamics of Mega Man. To verify that all evolved champion levels were beatable, they were uploaded to the Mega Man Maker servers, which do not allow levels to be uploaded unless a human player successfully beats them first.

5.3 Experimental Parameters

Levels were evolved using both the OneGAN and MultiGAN approach using NSGA-II with a population size of $\mu = \lambda = 100$ for 300 generations. Preliminary experiments indicated no significant improvements after 300 generations. The evolution experiment was repeated with each pre-trained generator 30 times.

Evolved levels consisted of 10 segments each, unless generation terminated early (see Section 4.3). Since the GAN latent input size was 5, and each segment used 4 auxiliary variables for determining relative placement, the total length of each real-valued genome was $(5 + 4) \times 10 = 90$ variables in the range $[-1, 1]$. Each generation offspring were created using a 50% chance of single-point crossover. Whether crossover occurred or not, every offspring was mutated: each real-valued number in the vector had an independent 30% chance of polynomial mutation [4].

6 RESULTS

Levels produced by OneGAN and MultiGAN are compared quantitatively in terms of their solution path lengths and novelty, and qualitatively in terms of the final levels produced by each run.
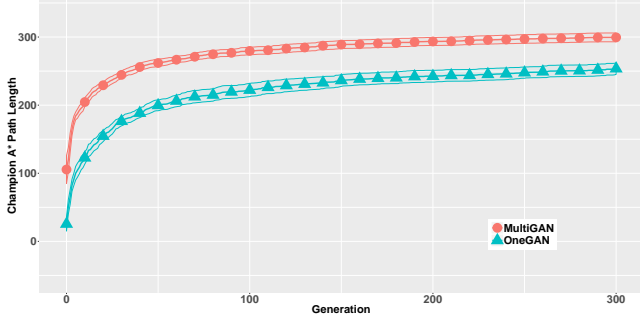
6.1 Quantitative Analysis

The A* path lengths are significantly longer ($p < 0.05$) with Multi-GAN than OneGAN (Fig. 3). The separation between methods is established early in evolution and maintained until the end. The difference in averages is approximately 50 tile steps throughout all of evolution. Because segments are 14×16 tiles, a difference of 50 means that OneGAN champions sometimes skip one or more segments, though it is also possible for paths to be lengthened with additional twists and turns inside individual segments.

Levels produced by the two approaches were also compared in terms of a novelty metric. Novelty can be defined in terms of an individual segment, a whole level, or an arbitrary collection of segments. Segment distance $d(x, y)$ between two segments x and y is defined as the number of positions in which their respective tile types differ, normalized to $[0, 1]$. Then, the novelty N of a segment x is defined as its average distance from all segments in some reference collection S:

$$N(x, S) = \frac{\sum_{y \in S} d(x, y)}{|S|} \qquad (1)$$

Figure 3: Average Champion A* Path Length Across Evolution. Plot of averages across 30 runs of OneGAN and MultiGAN of champion A* path lengths by generation. MultiGAN path lengths are approximately 50 steps longer than OneGAN paths throughout evolution. For context, recall that segments are 14×16 tiles.

Table 3: Average Level Novelty By Type

For each type of level, the LN of each level is calculated. The average LN score across N levels of the given type are presented below.

Type	N	Average ± StDev	Min	Max
VGLC	10	0.34±0.09	0.11	0.45
OneGAN	30	0.43±0.03	0.36	0.48
MultiGAN	30	0.41±0.02	0.34	0.46

The novelty of a level M is the average novelty of all segments it contains, where each segment's novelty is calculated with respect to the set of other segments in the level (excluding itself). The Level Novelty LN is:

$$LN(M) = \frac{\sum_{x \in M} N(x, M - \{x\})}{|M|} \qquad (2)$$

Within VGLC levels, segments are fairly uniform and less novel. Both GAN approaches produce levels where there is greater segment variety in each level compared to VGLC (Table 3). This suggests that there is more variety in GAN-generated levels than in the original game, but it may also suggest less consistent style.

Table 4 indicates that there are not that many repeated segments within the individual data sets for a given type of level. MultiGAN repeats segments more than OneGAN (fewer unique segments), likely because of the lack of training samples in corner segment GANs. In fact, Table 5 analyzes the distinct segments and novelty scores of each corner GAN from MultiGAN, showing that these corner GANs produced less novel results. However, despite some repetition in the corners, the overall novelty scores of OneGAN and MultiGAN are slightly more than VGLC.

6.2 Qualitative Analysis

The specific levels talked about in this section can be found in Table 6. Evolved champions from all 30 runs with OneGAN and MultiGAN can be viewed online[4] and played with Mega Man Maker.

OneGAN levels were more confusing due to the lack of flow. There are many random pits that Mega Man cannot traverse without the platform gun (see Section 3). MultiGAN levels generally look more natural, meaning adjacent segments connect and flow better.

[4]https://southwestern.edu/~schrum2/SCOPE/megaman.php

Table 4: Distinct Segments By Type

For each level type, the total number of segments, number and percentage of unique segments within the collection (removing duplicates to get a set), average segment novelty with respect to the collection, and average segment novelty with respect to the set (without duplicates) are shown. MultiGAN has the lowest percentage of distinct segments, but is between VGLC and OneGAN in terms of novelty, whether sets or complete collections are used, though the comparative novelty scores are all close.

	Segments	Distinct Segments	Average Novelty All	Average Novelty Set
VGLC	178	159 (89.3%)	0.4390	0.4349
OneGAN	300	287 (95.7%)	0.4709	0.4637
MultiGAN	300	254 (84.7%)	0.4542	0.4483

Table 5: Distinct Corner Segments in MultiGAN

Shows number of segments in MultiGAN levels generated by each corner GAN, number that were distinct, average novelty of collections from each GAN, and average novelty across the distinct segments. Corner segments from the same GAN often differ by only a few tiles, which is why novelty scores are low despite the sometimes high percentage of distinct segments.

	Segments	Distinct Segments	Average Novelty All	Average Novelty Set
Lower Left	42	28 (66.7%)	0.2453	0.2689
Lower Right	25	20 (80.0%)	0.3537	0.3448
Upper Right	40	29 (72.5%)	0.2635	0.2770
Upper Left	25	22 (88.0%)	0.2875	0.2857

Different MultiGAN levels tend to have the same or nearly identical corner segments because of issues with novelty pointed out in Table 5. However, a typical 10-segment level often only has one or two representatives of each corner type, so repetition of corner segments within the same level is very rare.

As seen in **OneGAN15**, OneGAN levels tend to have entire segments that are unreachable or unnecessary. One water segment in the figure leads to a dead end, and the other is blocked off by the other segments. Note that despite being specifically evolved to maximize solution length and connectivity, OneGAN struggled to connect segments. Similarly, in **OneGAN0** there are two unused segments at the bottom. These segments are reachable, but not needed. Examples like these explain why OneGAN's solution path lengths are on average 50 tiles shorter than those of MultiGAN.

However, OneGAN solutions do sometimes successfully traverse all segments, as seen in **OneGAN8**. However, even this level has large sections in the lower left and upper mid section that are not traversed, which further explains the shorter solution paths compared to MultiGAN.

In contrast, MultiGAN levels made better use of their allotment of 10 segments. In **MultiGAN1**, the average portion of each segment occupied by the solution path is roughly 12%, whereas in **OneGAN15** the portion is only 8%. Every ladder in the level is part of the solution path, suggesting a more efficient use of space and intelligent placement. In fact, the left-most side of **MultiGAN27** even presents an example of ladders between distinct segments perfectly aligning. **MultiGAN1** also effectively paired moving platforms with hazard spikes, which is an interesting challenge. MultiGAN levels were good at placing blocks in such a way that a simple
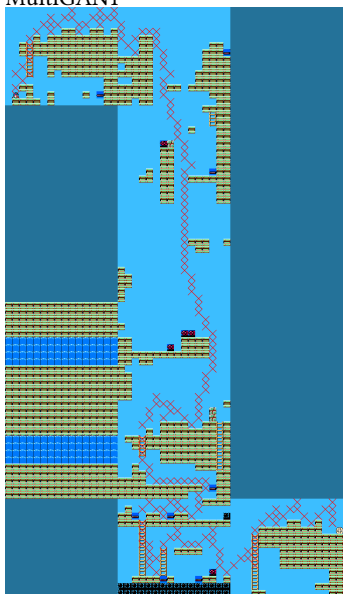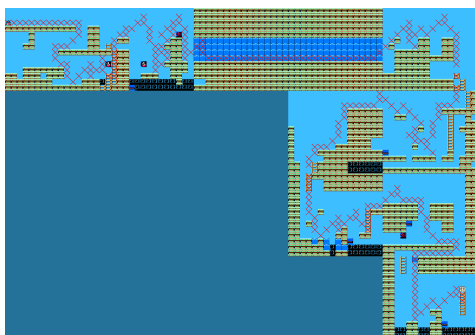
Table 6: Example Generated Levels.

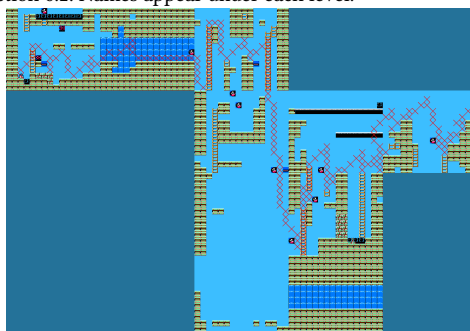Each is discussed in detail in Section 6.2. Names appear under each level.

OneGAN0

MultiGAN1
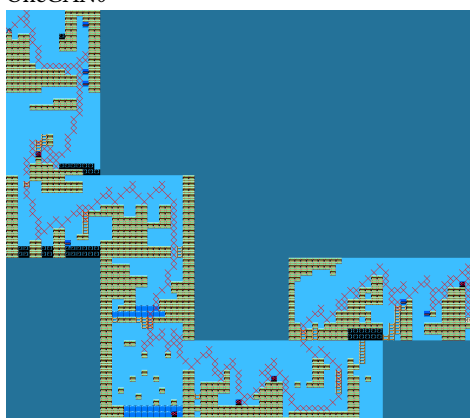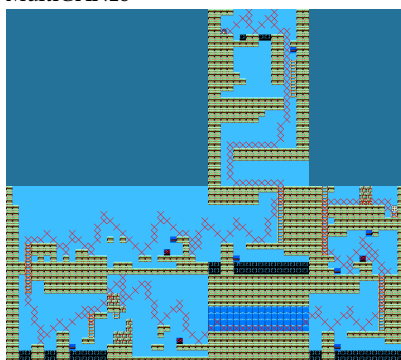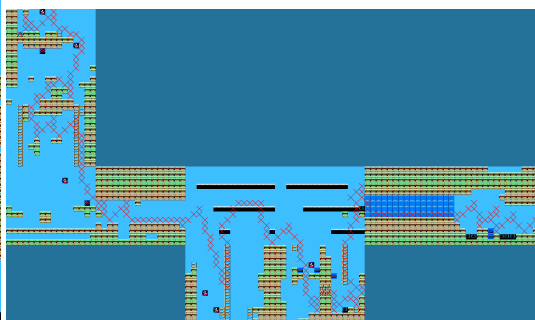

MultiGAN25

OneGAN15

MultiGAN27

MultiGAN7

OneGAN8

miscalculated jump could result in death, as in the original Mega Man, which is known for frustrating platforming challenges.

MultiGAN levels generally follow a snaking pattern not only in segment placement, but also in the solution path, whereas solution paths for OneGAN levels were more direct. Though MultiGAN does occasionally struggle with unnecessary block placement, such as in **MultiGAN25** with the lower left corner, the solution paths are both longer and more challenging to traverse than in OneGAN.

Human subjects confirmed some of these observations.

7 HUMAN SUBJECT STUDY

This section briefly describes a human subject study comparing OneGAN and MultiGAN levels, and the results of the study.

7.1 Human Subject Study Procedure

The study was advertised globally on social media and forums, but most of the 20 participants were undergraduate students from Southwestern University. Response was limited because participants seemed unwilling or unable to install and create an account for Mega Man Maker, which was required in order to participate.

Participants played a random evolved champion level of each type. Subjects were not informed how levels were generated, and were led to believe that some levels *might* be made by humans. Participants compared levels based on how difficult, fun, human-like, and interesting the designs were.

Mega Man Maker allows for a rich diversity of tiles that vary mainly in their appearance, but the tile types produced by the GAN (Table 1) are relatively plain. As a result, levels look simpler than typical human-designed levels. One study participant who had experience with Mega Man Maker pointed out that our levels would be impossible to make using the level editor because the editor blends different tiles from a given tile set to distinguish surfaces, corners, etc. As a result, this user could tell that our levels were made through direct manipulation of level text files.

Players had access to the platform-gun, allowing them to more easily traverse the more difficult, or otherwise impossible, jumping puzzles. Participants were asked if they thought the platform-gun was required to beat the level. Even with the platform-gun, some players did not beat both levels. They were encouraged, though not required, to make several attempts at each.

7.2 Human Subject Study Results

Quantitative results from the human subject study are in Table 7, including a statistical analysis.

Because it was difficult to find expert Mega Man players, several participants struggled to complete the levels. Only 14 participants completed the OneGAN level, whereas 17 completed the MultiGAN level. Additionally, 11 thought the platform-gun was required to beat the OneGAN level, whereas only 8 believed this of the Multi-GAN level. Needing the platform-gun could indicate bad design, as most actual levels can be traversed by normal jumping.

Most participants found their OneGAN level harder than their MultiGAN level. One respondent who said the OneGAN level was harder noted that the ladder placement "made no sense/led to nowhere/death screen." Another respondent said the OneGAN level "seemed hard in a random and not very well thought-out way." Yet

Table 7: Human Subject Study Results

Shows the data associated with the Human Subject Study that was conducted. Under Type, "y/n" means participants could answer *yes* or *no* for each individual level, and "e/o" means the participant had to pick *either* one level *or* the other. Responses to "e/o" questions were compared using one-sided binomial tests whose approximate p values are shown in the last column, where **bold** values indicate a significant difference ($p < 0.05$)

Question	Type	OneGAN	MultiGAN	p
Completed?	y/n	14	17	N/A
Platform-Gun Required?	y/n	11	8	N/A
Created by a Human?	y/n	4	13	N/A
Harder?	e/o	14	6	0.05766
More Fun?	e/o	2	18	**0.0002012**
More Human-Like?	e/o	5	15	**0.02069**
More Interesting?	e/o	7	13	0.1316

another said they could not see where they were falling and that it was difficult to "time [their] falls." A different participant said that OneGAN was harder because it "seemed to force the player to make a leap of faith at the start" which led to confusion.

A significant number of participants found the MultiGAN level more fun ($p < 0.05$). One participant said that the level "'knew' what [they were] going to do and could make things deliberately harder for [them] in an intelligent way," whereas the OneGAN level was "hard in a seemingly unintentional way." Another participant said that they "liked how much longer [MultiGAN] was. There were many places [they] could go." However, two participants said that neither were particularly enjoyable, and another participant said that the MultiGAN level "felt more like [they] were supposed to lose, where [the OneGAN level]...wanted to be beaten." For that particular user, the OneGAN level had five segments where the player could fall down to quickly and easily traverse a large portion of the level, whereas the SevenGAN level had more enemies, harder jumping puzzles, and a longer solution path. Such comments also explain why a majority of users thought MultiGAN levels were more interesting than OneGAN levels.

A significant number of participants ($p < 0.05$) thought the MultiGAN level's design was more human-like than OneGAN's, and 13 thought the MultiGAN level was created by a human in comparison to OneGAN's 4, though some thought correctly that neither was designed by a human. One participant said the OneGAN level was not created by a human because it spawns the player next to enemies, and it "feels impossible to not take damage from them." Another participant said that the OneGAN level "seemed like it was entirely random" and that it made it "frustrating." Yet another said that the MultiGAN level was made by a human because it "had more repeated building elements" and that the OneGAN level had no such pattern, and would sometimes have a "solitary floating block." Finally, one participant said the OneGAN level "seemed more random" than MultiGAN, which led to them thinking that it was "made by a human trying to trick the player."

8 DISCUSSION AND FUTURE WORK

With OneGAN, the barriers between segments are unpredictable. The majority of the training data consists of horizontal segments, making such segments more likely. However, if Mega Man needs to

move up into a new segment, he could get stuck banging into the floor of the segment above him. Often, when Mega Man needs to move down to a new segment, the only way to do so is by falling into what looks like a pit trap, because the next segment was generated beneath a segment modelled on horizontal segments. Note that OneGAN does not appear to be suffering from mode collapse [23]; but it does struggle to pick the right type of segment for a given situation. Because there are higher odds of the next segment being unreachable, the only way to traverse some OneGAN levels is by side routes caused when the sequence of segments snakes back into being adjacent to an earlier segment. However, these side routes result in certain segments being skipped in the solution path.

MultiGAN does not have this problem. When an upward segment is needed, it can be generated reliably. Everything generated by the Up GAN will provide ladders and/or platforms that make such movement possible. Similarly, when movement transitions from horizontal to vertical, the MultiGAN will use an appropriate corner GAN. However, corner GANs are trained on significantly fewer segments than their horizontal and vertical counterparts, causing a lack of diversity in generated corner segments (Table 5). This lack of diversity will generally not be noticed in any individual MultiGAN level, but repetition of specific corner segments can be seen across levels. Also, those familiar with the original game may notice the duplication of certain corner segments from Mega Man.

Both OneGAN and MultiGAN sometimes produce plain hallways filled with water. This segment is a reproduction of a segment that occurs repeatedly in Level 9, which consists of a long boring hallway filled with water. In the context of the original game, this scenario is an interesting departure from the norm, but the appearance of this segment in levels produced by GANs is usually out of place. In fact, the handling of water by both approaches can lead to unusual segments on occasion, indicating that some special handling of segments containing water may be necessary. However, despite the lack of continuity when handling water segments, LVE resulted in other types of segments derived from the diverse training set fitting well together.

Conditional Generative Adversarial Networks (CGANs [13]) could serve as an alternative to using multiple GANs. A CGAN is conditioned on some additional input, allowing it to produce output of a desired type on demand, thus eliminating the need for multiple GANs. However, the imbalance of training data would be a more serious issue for CGANs than it is for MultiGAN. The data disparity between horizontal/corner segments is nearly 150 to 1.

Larger training sets could solve these problems. There are many possible sources for Mega Man levels, such as the many other games in the Mega Man franchise. Data for games beyond Mega Man 1 are not in VGLC, but such a data set could be constructed. A more readily available source of levels is on the Mega Man Maker servers. These can be freely downloaded, and if properly simplified, could be converted into a format usable for training. With a large enough training set, MultiGAN should be able to produce greater diversity in its corner segments. However, more data in general means there will still be an imbalance with respect to corner segments, so simply using more data still might not make CGANs easier to apply.

In lieu of more training data, better corner diversity could be achieved by being more permissive about what counts as a corner segment. Currently, a sequence of horizontal and vertical window slices lead into and out of every occurrence of a corner segment. However, the three or four window slices surrounding each corner segment could potentially serve as viable corner segments as well. For example, if a corner segment has a floor section that is four tiles deep, then sliding the window up three tiles still leaves a floor at the bottom of the screen. Ladders and platforms for vertical movement would remain. Including such segments in the training sets for corners would increase their sizes without resulting in corner segments that could not be traversed. These new members of the corner sets could also be removed from horizontal and vertical data sets, slightly improving the balance of data.

A larger problem affecting OneGAN and, to a lesser extent, MultiGAN, is the issue of continuity between adjacent segments. The levels in the training set are all stylistically different, so if one simply stitched together a series of horizontal segments from different levels, the result would likely not be cohesive. To some extent, use of proper fitness functions during evolution creates some cohesion, but it would be easier if evolution did not need to make up for shortcomings in the GAN and genotype encoding.

A recently developed approach that could address this issue is CPPN2GAN [19], which uses Compositional Pattern Producing Networks (CPPNs [20]) to generate GAN latent inputs. CPPN outputs vary gradually as a function of segment location within the level, and since similar latent vectors result in similar GAN outputs, adjacent segments should be linked in a more cohesive way. This approach has the added benefit of scale, because levels of arbitrary size can be generated by a compact CPPN.

Because MultiGAN produced less variety in its corner segments, its levels were slightly less diverse than OneGAN's. Having better training data could fix this, but another way to increase diversity is to explicitly favor it during evolution. Quality diversity algorithms like MAP-Elites [14] could help in discovering such diverse levels, as has already been done in other GAN-based approaches [6, 19, 21].

9 CONCLUSION

When using GANs to generate levels with various segment types, it helps to use multiple GANs. Doing so preserves the structure of each segment type. In Mega Man, each segment type affects how adjacent segments connect. If poorly connected segments are generated, as with OneGAN, then Mega Man cannot properly traverse the entire level. MultiGAN is proposed to allow generation of the right type of segment whenever needed. MultiGAN was effective in producing levels with longer solution paths going through all available segments, in a way reminiscent of human-designed levels from the original game. In fact, a significant number of human subjects confirmed that MultiGAN levels had more human-like design and were more fun, in contrast to OneGAN levels which often had unusual barriers, unreachable segments, and overall stranger level design. MultiGAN shows promise for the generation of more complex, challenging, and cohesive levels, and future extensions to the approach should result in more diverse level designs as well.

ACKNOWLEDGMENTS

REFERENCES

[1] Martin Arjovsky, Soumith Chintala, and Léon Bottou. 2017. Wasserstein Generative Adversarial Networks. In *International Conference on Machine Learning*.

[2] Philip Bontrager, Wending Lin, Julian Togelius, and Sebastian Risi. 2018. Deep Interactive Evolution. *European Conference on the Applications of Evolutionary Computation (EvoApplications)*.

[3] Philip Bontrager, Julian Togelius, and Nasir Memon. 2017. DeepMasterPrint: Generating Fingerprints for Presentation Attacks. *arXiv:1705.07386* (2017).

[4] Kalyanmoy Deb and Ram Bhushan Agrawal. 1995. Simulated Binary Crossover For Continuous Search Space. *Complex Systems* 9, 2 (1995), 115–148.

[5] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. 2002. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* (2002).

[6] Matthew C. Fontaine, Ruilin Liu, Ahmed Khalifa, Julian Togelius, Amy K. Hoover, and Stefanos Nikolaidis. 2020. Illuminating Mario Scenes in the Latent Space of a Generative Adversarial Network. arXiv:2007.05674 [cs.AI]

[7] Edoardo Giacomello, Pier Luca Lanzi, and Daniele Loiacono. 2019. Searching the Latent Space of a Generative Adversarial Network to Generate DOOM Levels. In *Conference on Games*. IEEE, 1–8.

[8] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems*. 2672–2680.

[9] Jake Gutierrez and Jacob Schrum. 2020. Generative Adversarial Network Rooms in Generative Graph Grammar Dungeons for The Legend of Zelda. In *Congress on Evolutionary Computation*. IEEE.

[10] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. 2018. Progressive Growing of GANs for Improved Quality, Stability, and Variation. In *International Conference on Learning Representations*.

[11] Vikram Kumaran, Bradford W. Mott, and James C. Lester. 2020. Generating Game Levels for Multiple Distinct Games with a Common Latent Space. In *Artificial Intelligence and Interactive Digital Entertainment*. AAAI.

[12] Jialin Liu, Sam Snodgrass, Ahmed Khalifa, Sebastian Risi, Georgios N. Yannakakis, and Julian Togelius. 2020. Deep Learning for Procedural Content Generation. *Neural Computing and Applications* (2020).

[13] Mehdi Mirza and Simon Osindero. 2014. Conditional Generative Adversarial Nets. arXiv:1411.1784 [cs.LG]

[14] Jean-Baptiste Mouret and Jeff Clune. 2015. Illuminating Search Spaces by Mapping Elites. *arXiv:1504.04909* (2015).

[15] Kyungjin Park, Bradford W. Mott, Wookhee Min, Kristy Elizabeth Boyer, Eric N. Wiebe, and James C. Lester. 2019. Generating Educational Game Levels with Multistep Deep Convolutional Generative Adversarial Networks. In *Conference on Games*. IEEE.

[16] D. Perez-Liebana, S. Samothrakis, J. Togelius, T. Schaul, S. M. Lucas, A. Couetoux, J. Lee, C. U. Lim, and T. Thompson. 2016. The 2014 General Video Game Playing Competition. *Transactions on Computational Intelligence and AI in Games* (2016).

[17] Anurag Sarkar and Seth Cooper. 2020. Sequential Segment-Based Level Generation and Blending Using Variational Autoencoders. In *Foundations of Digital Games*. ACM.

[18] Jacob Schrum, Jake Gutierrez, Vanessa Volz, Jialin Liu, Simon Lucas, and Sebastian Risi. 2020. Interactive Evolution and Exploration Within Latent Level-Design Space of Generative Adversarial Networks. In *Genetic and Evolutionary Computation Conference*. ACM.

[19] Jacob Schrum, Vanessa Volz, and Sebastian Risi. 2020. CPPN2GAN: Combining Compositional Pattern Producing Networks and GANs for Large-scale Pattern Generation. In *Genetic and Evolutionary Computation Conference*. ACM.

[20] Kenneth O. Stanley. 2007. Compositional Pattern Producing Networks: A Novel Abstraction of Development. *Genetic Programming and Evolvable Machines* 8, 2 (2007), 131–162.

[21] Kirby Steckel and Jacob Schrum. 2021. Illuminating the Space of Beatable Lode Runner Levels Produced By Various Generative Adversarial Networks. arXiv:2101.07868 [cs.LG]

[22] Adam James Summerville, Sam Snodgrass, Michael Mateas, and Santiago Ontañón. 2016. The VGLC: The Video Game Level Corpus. In *Procedural Content Generation in Games*. ACM.

[23] Hoang Thanh-Tung and Truyen Tran. 2020. Catastrophic Forgetting and Mode Collapse in GANs. In *International Joint Conference on Neural Networks*. IEEE.

[24] Julian Togelius, Emil Kastbjerg, David Schedl, and Georgios N Yannakakis. 2011. What is Procedural Content Generation? Mario on the Borderline. In *International Workshop on Procedural Content Generation in Games*. ACM.

[25] Ruben Rodriguez Torrado, Ahmed Khalifa, Michael Cerny Green, Niels Justesen, Sebastian Risi, and Julian Togelius. 2020. Bootstrapping Conditional GANs for Video Game Level Generation. In *Conference on Games*. IEEE.

[26] Vanessa Volz, Jacob Schrum, Jialin Liu, Simon M. Lucas, Adam M. Smith, and Sebastian Risi. 2018. Evolving Mario Levels in the Latent Space of a Deep Convolutional Generative Adversarial Network. In *Genetic and Evolutionary Computation Conference* (Kyoto, Japan). ACM.