Procedural Content Generation of Puzzle Games using Conditional Generative Adversarial Networks

ANDREAS HALD, IT University of Copenhagen, Denmark
JENS STRUCKMANN HANSEN, IT University of Copenhagen, Denmark
JEPPE KRISTENSEN, IT University of Copenhagen, Denmark
PAOLO BURELLI, IT University of Copenhagen, Denmark

In this article, we present an experimental approach to using parameterized Generative Adversarial Networks (GANs) to produce levels for the puzzle game Lily's Garden[1]. We extract two condition-vectors from the real levels in an effort to control the details of the GAN's outputs.
While the GANs performs well in approximating the first condition (map-shape), they struggle to approximate the second condition (piece distribution). We hypothesize that this might be improved by trying out alternative architectures for both the Generator and Discriminator of the GANs.

Additional Key Words and Phrases: Procedural Content Generation, Conditional Generative Adversarial Networks, Puzzle Games

ACM Reference Format:
Andreas Hald, Jens Struckmann Hansen, Jeppe Kristensen, and Paolo Burelli. 2020. Procedural Content Generation of Puzzle Games using Conditional Generative Adversarial Networks. In *Malta '20: 11th Workshop On Procedural Content Generation (PCG 2020), September 15–18, 2020, Bugibba, Malta.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

PCG can be defined as the generation of content for a game by algorithms which require little to no supervision by a game-designer [13, p. 1]. As content production is an often tedious and time-consuming task, game-companies are increasingly utilizing PCG in an effort to alleviate some of the pressure of having human designers create all of the content within games. But content production often relies on domain specific knowledge - in order to create levels that are playable and continually enjoyable as they enter into the customers overall experience of the game - and in light of this requirement it is often challenging to automate the entirety of the content production with algorithms. However, this is exactly what researchers are attempting in a relatively new paradigm in PCG called Procedural Content Generation via Machine Learning (PCGML).

In recent years we have seen studies in PCGML using Generative Adversarial Networks (GAN). However, GANs are generally not great at upholding the constraints needed for necessary content used in games [19]. Previous studies have used different methods to overcome the challenges of keeping the content functional for games by using evolutionary algorithms and capturing information about spatial relationships in levels [19, 20].

[1] https://tactilegames.com/lilys-garden/

Malta '20, September 15–18, 2020, Malta

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

In this study we are drawing inspiration from mixed-initiative generation where humans and computers interact to pull the content in a certain direction, specifically computer-aided design tools like Sentient Sketchbook [8, 13].

A natural extension to our work could therefore be to create a tool that can aid the content producers in speeding up the process of generating game-levels for the puzzle-game Lily's Garden. To do this we present a preliminary study of using Conditional Generative Adversarial Networks to generate a draft of levels based on the variables of the game that we believe to be domain specific.

2 RELATED WORK

2.1 Procedural Content Generation

Procedural Content Generation (PCG) focuses on using algorithms to produce various types of content that is used for games [13, p. 1]. Content include levels, quests, textures and the PCG methods used are usually specialized in doing one particular type of task, but Shaker et. al have also suggested multi-content PCG, as an alternative path [13, p. 5].

Throughout the existence of PCG the motivation behind using it has changed. In the early days of video games PCs didn't have a lot of storage space, so instead of saving different game layouts, they could be randomly generated [13, p. 4]. Today, game development for AAA games has become more expansive which requires more designers and time. As such, if algorithms alleviate the need for increasing staff or help make designers more efficient it would certainly be beneficial, which is also true for smaller teams [13, p. 3]. Another aspect of today's use of PCG is the possibility of personalizing content for players to keep them engaged, which could span from different scenarios that a player enjoys, to the difficulty or patterns in a platform game that a player keeps going back to [14, 18].

In recent years a new paradigm in PCG has emerged, Procedural Content Generation via Machine Learning (PCGML) [17]. This paradigm separates itself from PCG in the sense that while PCG approaches use machine learning models, PCGML samples directly from the model, based on existing distributions of content [17]. Examples of this paradigm includes Snodgrass & Ontañón's work, in which they use Constrained Multi-Dimensional Markov Chains in order to control the generation of both levels for the games Super Mario Bros and Kid Icarus [15]. Another example is Sarkar et. al's. use of Variational Autoencoders for Controllable Level Blending between Games [12], Sarkars work is especially interesting in that they seek to optimize for multiple different features, which has a lot of commonality with the line of inquiry that we are pursuing.

This consequently is the paradigm of PCG that we are working under, with a focus on Generative Adversarial Networks which we will briefly describe next.

2.2 Generative Adversarial Networks

Generative Adversarial Networks (GAN) is a collection of architectures introduced by Ian Goodfellow et al. [4]. in 2014. The basic idea is that two neural networks, a generator and discriminator, are competing against each other in a mini-max game, where the generator tries to fool the discriminator into predicting its output as real, and the job of the discriminator is to distinguish the fake samples of the generator from real samples of data [3, 4]. The two models are trained simultaneously and theoretically the mini-max game should lead to convergence. However, there are multiple challenges when balancing the architecture which has been a focus point in the continued research on the subject [3]. Multiple improvements has been made to the GAN architecture, such as the Deep Convolutional GANs [10], which is broadly considered a general improvement on the original model.

2.3 PCGML and GAN

Since the GAN architecture was presented in 2014, we are aware of three articles using GANs to generate content for games [2, 19, 20]. Giacomello et al. (2018) use levels from the original DOOM by extracting topological features

Fig. 1. Lily's Garden level 108 at the initial state.

and level images to generate new levels with two different GANs-architectures. The purpose is to evaluate the networks performance in generating levels similar to human designed ones, with both training on level images and one conditioned on the topological features as well [2].

Volz et al. (2018) train a GAN on mario levels and uses an evolutionary algorithm on the latent space to improve properties relating to difficulty, and playability is also checked using a player agent [20].

Torrado et al. (2019) aim to mitigate common problems in PCG like data scarcity and generating functional content for games by implementing Self-attention to make sure that objects required for functionality (in their case a key and door) is to be found within a synthetic level, and adding playable examples of generated content to the training loop for their GAN [19].

Our approach uses some of the same ideas shown in the aforementioned articles, but we explore them in the context of possibly creating a mixed-initiative design-tool between GANs and game level designers at a later stage.

3 METHOD

In this section, we will first give the reader an impression of what the game *Lily's Garden* is about. Secondly we will describe the way that we represent the game levels as data to be used in GANs. Thirdly we discuss the important variables, specific to the puzzle-game genre, and finally we describe the GAN-algorithms that we have implemented in our project.

3.1 A Lily's Garden tutorial

Unique Pieces		*Clickable*
Color Island		*Clickable*
Blockers		*Non-Clickable*

Table 1. Table viewing a Term, its visualization, and whether or not it is clickable.

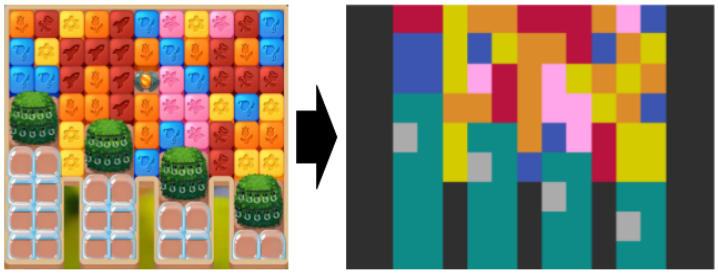

Fig. 2. Reduced Representation of level 108

Lily's Garden is a puzzle game where the player has to collect a certain amount of the *pieces* located on the game-board (see left column in fig. 1). Collecting pieces can be done by connecting two or more *clickable*-pieces of the same color (For example the 7 red connected pieces in the top left quadrant of fig. 1) which makes them clickable (referred to as *color-islands*). However, some pieces are not themselves clickable (referred to as *blockers*). This is for example the bushes in fig. 1 and in this case, the only way to collect them is by clicking on a color-island touching upon the non-clickable piece. While there are some other categories of pieces, the color-islands and blockers are the most important types and we will limit ourselves to focusing on those for the remainder of the article.

3.2 Data representation

3.2.1 The Representation Challenge: Within PCGML there isn't a general agreement on how to represent the game structure [17, p.11], but most examples we have seen seem to represent each distinct piece in a game-level by a distinct channel within a 3D matrix [2, 19, 20]. However, Lily's Garden has 56 unique pieces and vastly more combinations of unique pieces stacked upon each other. By comparison (and to our understanding) Volz et. al. [20] represented an entire Super Mario Bros level with just 10 unique piece representations.

3.2.2 Reducing the number of channels: In order to reduce the complexity of cleaning, visualizing and analysing the data, we therefore decided to reduce the data representation into 8 distinct piece-types: (1) Cell-layer, indicating the levels shape, and where other unique pieces are allowed to be placed. (2) Blocker-layer (3)-(8) color-layers 1-6. - We realize doing this is to reduce the problem we were originally trying to solve, but an important feature of the human-made levels are, that the main way to *start* a level is by clicking on a color-island and thus the basic but important feature of *startability* is maintained.

Fig. 2 shows the reduced representation which has some key differences from the original. The bubble pieces are not visible in our representation and the *bushes* are represented with 1 blocker piece, you might also notice that the piece placement is not exactly the same, this is because the game state are generated on different random seeds. The first two dimensions of the levels are originally (9,13), but we increase it to (9,15) by padding with empty values. This is done to be able to apply transposed convolutional layers in our generator, which makes it possible to go from (3,5) up to the (9,15) size. The third dimension is the number of unique pieces in our representation, which gives us a final multi-channel representation of 15x9x8.

3.3 Important variables

Every game has some defining variables which are important to the functionality of a level and further the player experience. In this section we will briefly mention those that we consider important in Lily's Garden.

3.3.1 Shape: Shape is important because it determines where pieces can spawn on the screen and further because it limits where the player can interact with the level.

Fig. 3. 8 randomly selected level-shapes from Lily's Garden, visualized as heatmaps.

It is for this reason that an entire unique piece-type in our data-representation is indicatory of the level shape (the cell-layer). Additionally, given the many varieties of shapes there exists within the human made levels (see fig. 3) we found it reasonable to assume that in a PCG-context, a variable that game-designers might want to control is the level-shape.

3.3.2 Piece Distribution. The distribution of pieces in a level is also an important variable. In fig. 1 for example, it is evident that some pieces, such as the red- and orange-cookies (henceforth referred to as clickable-types), are more dominating than the pink or the blockers underneath. In Lily's Garden there generally are a lot of unique pieces and while the most re-occurring pieces are the clickable-type, a level is often dominated by clickable- and some *particular* blocker types. While the clickable pieces are typically clustered into color-islands - with some additionally lonely pieces being scattered across the board - the blockers typically make up some local shapes within the global shape. Given their visual importance, as well as role in the functionality of a level, we also found the piece-distribution a plausible variable that game-designers might want to be able to control.

3.4 Train and test set

We had access to 655 unique levels of Lily's Garden. But to increase the amount of data even further, we flipped the levels horizontally, vertically and diagonally in each channel, giving us 2620 levels. We then used 85% of the levels for training and 7.5% (196 levels) for test and validation set respectively. While we do not use the validation set in this article, they are stored for possible future work.

3.5 GAN architectures

In this article, we have developed two types of GANs: 1. The *Wasserstein GAN using Gradient Penalty with Parametric Embeddings* (WGAN-GP-PE) and the *Conditional Wasserstein GAN using Gradient Penalty* (CWGAN-GP). The exact architecture of the generators and critics can be accessed here[2]. Let us unpack these models a few terms at a time:

3.5.1 Wasserstein and Gradient Penalty: As the name may suggest we utilize the wasserstein-loss function which was suggested by Arjovsky et. al. as a general improvement over cross-entropy and leads to more stable training of GANs [1]. Further we use *Gradient Penalty* as suggested by Gulrajani et. al. as a general improvement to the wasserstein-loss, stabilizing the gradients and theoretically approximating the real data distribution better [5]. In practice, this also proved to be the case for us.

3.5.2 Conditonal and Parametric Embeddings: Traditionally GANs [4] only get the random-vector $\mathbf{z}$ (drawn from some normal distribution) as an input-vector. However, if one wants to control the output of the generator towards producing specific types of images, one has to introduce conditions as suggested by Mirza et. al. [9]. In

[2]https://github.com/DresRumler/pcg-workshop-visualizations

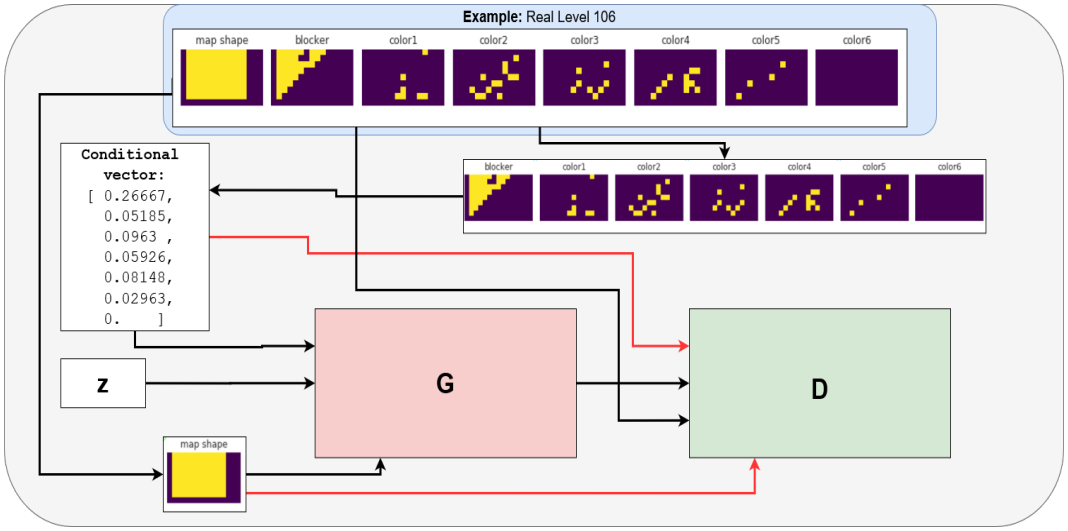

Fig. 4. WGAN-GP-PE and CWGAN-GP, the red lines are only included in the CWGAN-GP model. The level-shape and the piece-distribution of each level is extracted as conditions. The piece-distribution-vector is generated by counting the amount of 1's in the blocker- and color-channels and assigning each value to an index in the condition vector. Each number is then divided by 135 (total possible size of a level) to normalize the values between 0-1, indicating the proportion of the level that a particular channel inhabits.

our case, the conditions are the level-shape and piece-distribution, which we described in section 3.3. As can be seen in fig. 4 we use both variables as input to the model. In the case of the WGAN-GP-PE we only feed the generator the variables (the red-lines in fig. 4 are not included), where in the CWGAN-GP we also feed them to the discriminator (the red-lines in fig. 4 are included). This is the only difference between the models, but given that the discriminator is actually not taking the conditional vectors into consideration in the WGAN-GP-PE-model, we decided to define the conditional vectors in this case as *Parametric Embeddings*.

3.5.3 *Hyperparameters.*

- Loss-function: Wasserstein Loss.
- Critic trained 5 times for every 1 time Generator is trained.
- Optimizer: Adam with learning rate 0.0001, beta term 1 and 2 equal to 0.5 and 0.9 respectively.
- Batch-size: 32 levels.
- Epochs: 300

4 EXPERIMENTS

Most GAN-literature has concerned itself with producing realistic looking images of human-faces[3] and more generally objects from the ImageNet[4]. As such, the most used evaluation metrics, such as the Inception-score [11] and the Fréchet Inception Score [7] is also mainly useful in this context of measuring how realistic looking an image is.

To judge the quality of a game-level however - and specifically for Lily's Garden - is quite a different task and

[3]http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html
[4]http://image-net.org/

we have therefore run 7 distinct experiments which seeks to test the quality of our generators from numerous different angles.

4.1 Data distribution approximation of the generators

Firstly, we investigate how well our generators has approximated the data distribution of the training set, by looking at the proportion of pieces that the generated data sets and training set contain respectively. We have generated 250 samples for each level in the test set (of 196 levels) Which gives us a total of 49.000 synthetic levels for each of the generators.

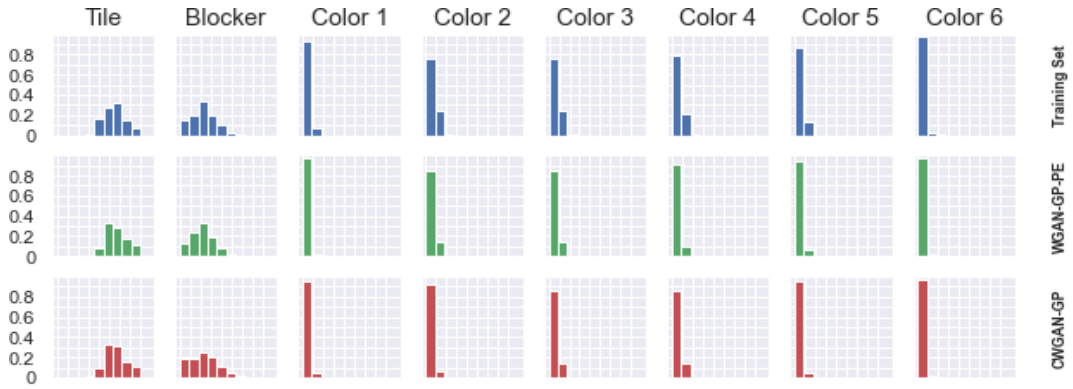

Fig. 5. Distributions of the Training Set, WGAN-GP-PE and CWGAN-GP respectively. X-axis ranges between 0 and 135 and Y-axis is the proportion of piece-amount that falls within a given range.

From fig. 5 and tab. 2, it is clear that both generators have approximated the training distribution pretty well with regards to the number of pieces that each cell-layer contains. However, the color layers seem to be on the low side of the training-set consistently, while the cell-layer is slightly over represented.

Table 2. 0.5'th quantile and standard error

Layer	Training Set	WGAN-GP-PE	CWGAN-GP
Cell	$81\pm^{17}_{14}$	$84\pm^{18}_{14}$	$84\pm^{17}_{15}$
Blocker	$34\pm^{17}_{19}$	$32\pm^{16}_{17}$	$33\pm^{22}_{22}$
Color	$0\pm^{11}_{0}$	$0\pm^{4}_{0}$	$0\pm^{9}_{0}$
Color 2	$9\pm^{6}_{9}$	$6\pm^{7}_{4}$	$5\pm^{6}_{3}$
Color 3	$9\pm^{6}_{5}$	$7\pm^{6}_{4}$	$7\pm^{6}_{4}$
Color 4	$9\pm^{5}_{5}$	$6\pm^{5}_{3}$	$7\pm^{6}_{4}$
Color 5	$6\pm^{6}_{6}$	$4\pm^{5}_{3}$	$4\pm^{5}_{3}$
Color 6	$0\pm^{5}_{0}$	$1\pm^{4}_{1}$	$0\pm^{3}_{0}$

The generators must be said to have come close to the training-data distribution with respect to the proportion of pieces, but whether this translates into levels that share commonalities to the training set is what we will turn our attention to now.

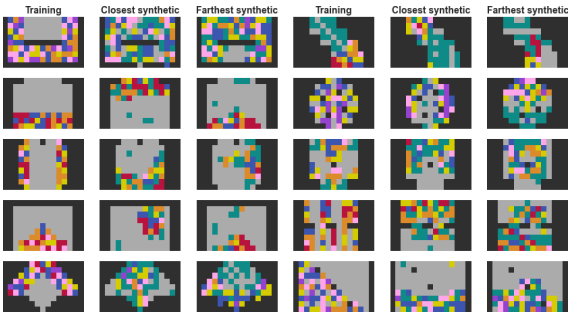

Fig. 6. WGAN-GP-PE

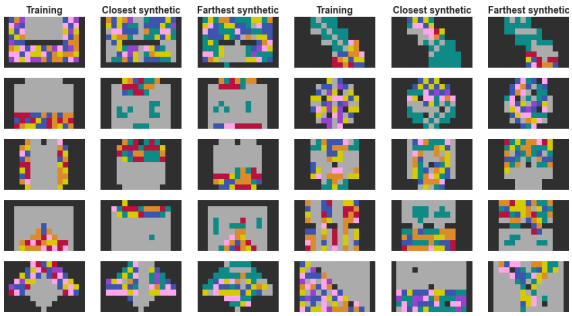

Fig. 7. CWGAN

Fig. 8. Test levels compared to the closest- and farthest synthetic level of the Generators in Wasserstein distance.

4.2 Visual Quality

In this experiment, we compare the training set levels, to the closest and farthest synthetic levels measured in wasserstein-distance, the same metric used for the generators cost-function.

4.2.1 Piece distributions. The most obvious difference between the training- and synthetic-set examples in fig. 6 & 7, is how representative *empty* board pieces (represented with the green color) is. In the training-set examples, only 2 out of 10 levels contains empty pieces, whereas every synthetic-level contains some empty pieces. Considering that both generators has been trained on the piece-distribution vectors of the training-set, the presence of empty pieces in every synthetic level, suggests that the generators has failed to utilize the piece-distribution vectors as we intended. We will return to this observation in section 4.4.

4.2.2 Symmetry. Another important aspect of the training levels, seems to be the symmetric smaller shapes within the level-shapes. Once again, both the generators seems to underperform in comparison to the training-set. However, the CWGAN-GP does seem to be a general improvement over the WGAN-GP-PE in this respect. A statement we base on the observations that: the synthetic levels of the CWGAN-GP has more clear-cut vertical and horizontal borders between the blocker-and color pieces. Further, the CWGAN-GP produces these pyramidic-shapes (see row 4, column3 (refered to as (4,3), and (5,6) in fig. 7, which seems to suggest that the model has also adapted well to diagonal lines (see fig. 7).

4.3 Effect of level-shape-conditional vector - Likely Playable

In this experiment, we test how well our models are reproducing the level-shapes it is given as inputs. To do this, we extract the cell-layer of the produced levels and compare them to the original input level-shape.

Fig. 9. A test-set *real* level-shape compared to the level-shape of 4 versions of the same synthetic levels of our CWGAN-GP

From fig. 9 it is clear that our CWGAN-GP approximates the real level shape quite well, albeit not perfectly. Notice that we have not rounded the synthetic values, and thus the green pieces in the images indicate where our

CWGAN-GP is uncertain of whether to place a piece. While this mean our models are not perfect, it is worth noting how many of the pieces that the network is absolutely certain (1 on a scale of -1 to 1) should be placed in the spot. But how far from perfect are our models generally? To test this we generate 250 levels for each level-shape and condition-vector. We then count the number of times a level has failed and succeeded to fill in a desired spot (underfilled/overfilled) and report the average. The results are summarized in table 3.

Table 3. Average under and overfilling of the generators levelshapes.

	WGAN-GP-PE	CWGAN-GP
Avg. underfilled	1.67	0.18
Avg. overfilled	2.07	0.085

Table 3 shows us that the CWGAN-GP approximates the level-shape-condition much better than the WGAN-GP-PE, which indicates that feeding the level-shape conditional vector to the discriminator as well, has helped the generator in better approximating the real shape. Nevertheless, 18% of the CWGAN-GP levels are underfilled and 8.5% is overfilled and as such there is room for improvement.

4.4 Effect of piece-distribution-conditional vector

In this experiment we test how well our models utilize the piece-distribution-vector. Recall that we feed a conditional-vector of size 7 (blocker- and color distributions) to the generator in both models and also the discriminator in our CWGAN-GP model. To test how it adjusts to the information in the vector, we will produce 100 new levels for each level in the test-set. We will then retrieve the distribution-vector for each of the produced levels and subtract the real distribution vector from the derived distributions-vectors. Ideally, the error should be close to 0, because this would mean that our models are recreating the distributions perfectly.

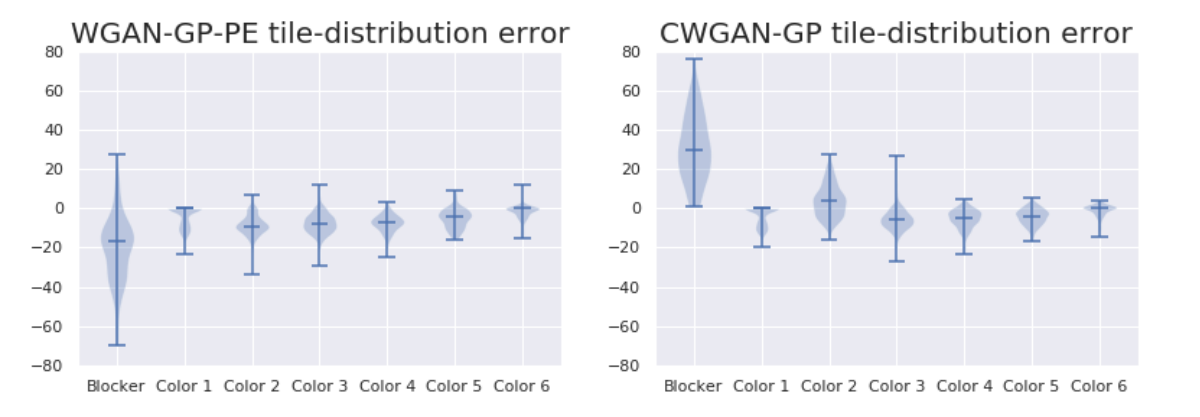

Fig. 10. Violinplots of each index in the distribution-vector, showing the error between real- and vectors derived from generated levels.

From the violinplots in fig. 10 it does not seem like the models have approximated the conditional vector. However, we hypothesize that this might have to do with the fact that we reduced a lot of unique pieces into the single piece *blocker* and thus our models might have learned that there are generally a lot of blocker-pieces, which it tries to distribute on every generated level.

4.5 Testing Color-Islands

We test *Color-islands* because it is indicative of whether there is anything clickable on a level to begin with and thus whether the level is *startable*. We test this by producing 250 generated levels for each level in the test-set (a batch), and we then count the number of color islands within each level.

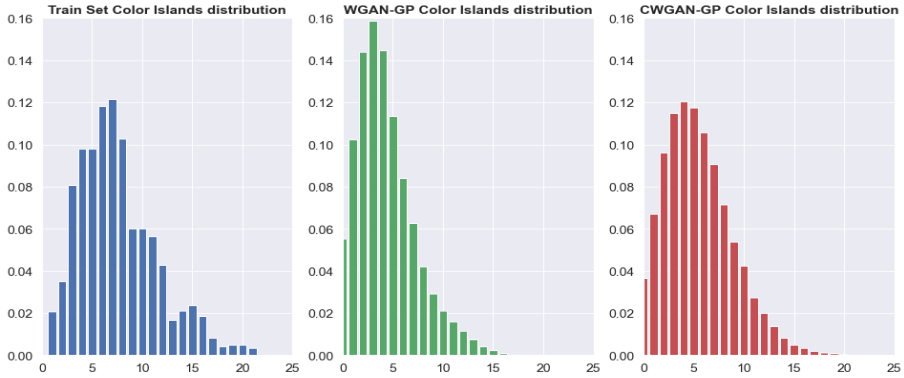

Fig. 11. Bar Graphs showing the distribution of color-islands in train-set, WGAN-GP-PE- and CWGAN-GP synthetic-set respectively.

Fig. 11, shows that both the WGAN-GP-PE and CWGAN-GP has approximated the distribution of training-set closesly, although with some important differences.

4.5.1 0 color-islands. From fig. 11 it is clear that the minimum amount of color islands in a training-set level is 1. By contrast, the synthetic levels of the WGAN-GP-PE contains ca. 5.5% levels with no color islands, and the CWGAN-GP contains a little less than 4%. This effectively means that the generators both generate levels that are *unstartable* and thus by extension *unplayable*.

4.5.2 Largest proportions of color-islands. Both the generators tends to produce less color-islands than are in the training set. For the WGAN-GP-PE ca. 16% of the levels has 3 color islands in them, but for the CWGAN-GP the highest proportion of levels are 12% at 4 color islands. By contrast, the largest proportion (ca. 12%) of levels in the training set has 7 color-islands in them.

4.6 Testing Broken pieces

We test *Broken pieces* by seeing if our model places any blocker- or color-piece outside the perimeter of the generated cell-layer. Like the preceding tests, this is done by producing batches of 250 generated levels and seeing how many of the levels within a batch are *broken*.

Our WGAN-GP-PE produces 46% levels that are not broken from the beginning, while our CWGAN-GP produces 66% (see fig. 12). Evidently our generators struggles more with this test than the color-island test.

Interestingly, while we found that the piece-distribution vector did not contribute much to the actual distribution of tiles in the synthetic levels (see sec. 4.4), the piece-distribution vector does seem to help improve the CWGAN-GP to generating 66% non-broken levels from the beginning. Recall that the main difference between the WGAN-GP-PE and the CWGAN-GP, is that the discriminator network is also fed the piece-distribution vector in the CWGAN-GP. Thus while the piece-distribution vector is not necessarily working the way we intended, it does seem to effect the quality of the synthetic levels, when the discriminator also gets to take it into account.

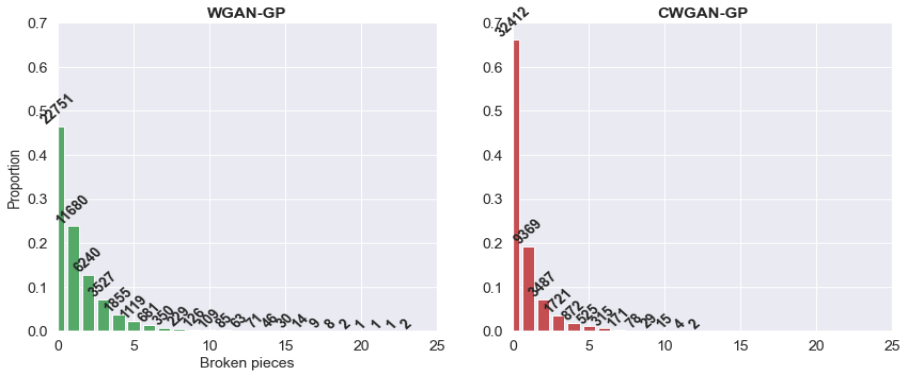

Fig. 12. Bargraphs showing the proportion of n-broken pieces in the 49000 synthetic levels generated for WGAN-GP-PE and CWGAN-GP respectively.

4.7 Expressive Range

In this section, we test the expressive range [16] of the generators, in comparison to training set in two different experiments.

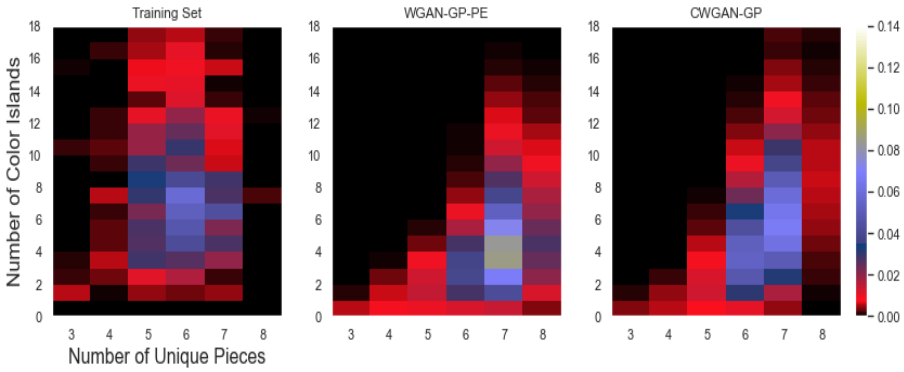

Fig. 13. 2D Heatmaps, showing the expressive ranges of number of unique pieces and number color islands within the train-WGAN-GP-PE- and CWGAN-GP-set of levels respectively.

Firstly, we test the generators expressive ability over the number of color islands and unique pieces in a level. Evidently, from fig. 13, both generators tend to overestimate the number of unique pieces in a level, but the WGAN-GP-PE more so than the CWGAN-GP. The training-set is generally much more spread across the middle of the heatmap, than the generators, which tend towards the lower right corner with all colors represented but fewer color islands. In other words, the training set usually has 5-7 unique pieces, whereas both generators typically has 6-8, although the CWGAN-GP once again is closer to the training-set than the WGAN-GP-PE.

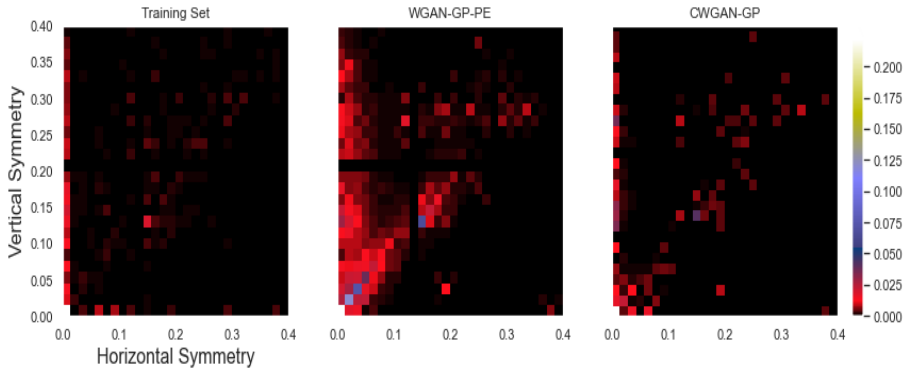

Fig. 14. 2D Heatmaps, showing the expressive ranges of the horizontal- and vertical symmetry within the train- WGAN-GP-PE and CWGAN-GP-set of levels respectively.

Secondly, we test the generators ability to create symmetrical levels by flipping the level-shapes (see fig. 14), first in the middle horizontal axis (X-coordinate) and secondly on the vertical (Y-coordinate). We then measure the hamming-distance [6] and use this as an indicator of how symmetrical a level is. Thus, if a level is both symmetrical on the X- and Y-axis, the distance is zero. Interestingly, the CWGAN-GP seems to adapt much better to the training-set, than do the WGAN-GP-PE. This is especially so since more than 20% (the colormap indicates the proportions) has a hamming-distance of 0, both on the X- and Y-axis.

5 DISCUSSION AND FUTURE WORK

Summarizing our results it seems our models perform well in approximating the level-shape input, but badly with respect to the distribution vector. Further, they perform well in the *color island*-test but struggles with the *broken piece*-test. Generally it must also be said that our CWGAN-GP outperforms the WGAN-GP-PE and going forward it is likely more rewardable to focus our attention towards the CWGAN-GP architecture. Let us look at the tests for condition-vectors and color-islands/broken pieces separately:

5.1 The condition vectors

In our generator we concatenate the **z**-vector and the **piece-distribution** vector. We then lead the concatenated result through some densely connected layers, before reshaping it into a (9, 15, 7) matrix, representative of the blocker- and color-layers in a level. Finally we concatenate the level-shape and the (9, 15, 7) matrix together. While we're not certain that the concatenation of the **z**-vector and **piece-distribution**-vector is the problem, it certainly hasn't proved to be a solution either. Conversely our model is adapting quite well to the level-shape, but this is also an input that can be fed in and remain non-altered before we apply convolutions, because it already has the (9, 15, 1)-shape. By contrast our **z**- and **piece-distribution**-vector has to be led through some densely connected layers in order to scale them up and into the (9, 15, 7) matrix. This could incentivize a look into other forms of representation of conditions with the closest example being Giacomello et al. (2018) which extracts a feature topology from DOOM levels, and in their case the model responded positively to this addition [2].
A possible improvement to the distribution-condition-vector might be to use some arithmetic operations on each of the layers, thus more directly connecting each cell in the condition-vector to the specific layer that it is meant to affect. This operation could be done once or on numerous occasions throughout the convolutions, so as to impose the importance of the distribution-condition-vector more explicitly.
A completely other avenue of inquiry, that we would also like to focus on in the future, is to focus on feedback

from the intended users to understand whether our current idea of parameters should change in favour of other parameters. This would greatly improve the likelihood of integrating GANs into mixed-initiative tools that are actually likely to work for the intended users.

5.2 Color islands and broken pieces:

Testing for color-islands and broken pieces also give two very differing results. But it is worth considering the nature of these tests: Testing color-islands is essentially a question of finding just 1 color-layer in which 2 pieces are located next to each other. Likewise for broken pieces just 1 piece has to be out of place, but where the color-islands counts as a passing grade, the broken pieces count as a failing. In some sense they can be seen as occupying two extremes of a spectrum in which we could create others tests and it would be instructive to try and formulate some tests that better breach this divide. With our current representation we are also missing an aspect of PCG regarding *broken-pieces* which goes beyond what we are currently testing. In "Evolving Mario Levels in the Latent Space of a Deep Convolutional Generative Adversarial Network" by Volz et al. they present the prevalence of broken tubes in their generated content for Mario, but in our content this simply isn't possible with our current representation but we presume that we will encounter the same issue once we introduce 2x2 pieces (such as the bushes in fig. 1) and this consequently will also be something that needs attention in the future [20].

6 CONCLUSION

In this paper, we presented two GAN-architectures (WGAN-GP-PE and CWGAN-GP) for producing simplified levels of the puzzle-game Lily's Garden. To test the synthetic levels generated by the GANs, we test its ability to create levels, by testing whether the GANs are reproducing the desired level-shape we feed them correctly, as well as the piece-distribution. Both GANs reproduce the level-shape well, although the CWGAN-GP seems to be performing best, while they both fail to pick up our intended meaning for the piece-distribution-vector. We further tested the *functionality* of the produced levels, by testing whether the models are producing levels with at least one *color-island* in them and further whether the GANs produces any *broken*-pieces. Both models perform well when tested for color islands, although the CWGAN-GP once again seems to be performing best. Conversely both models struggles when tested for broken pieces, although once again the CWGAN-GP seems to be performing better than the WGAN-GP-PE. In the future, it is therefore plausible that it is more feasible to continue with the CWGAN-GP model. However, before too much time is spend on optimizing for the variables we chose in this paper, it is worth investigating whether the condition-vectors we chose are also what game-designers might actually want or if entirely other variables might be of greater importance.

REFERENCES

[1] Martin Arjovsky, Soumith Chintala, and Léon Bottou. 2017. Wasserstein GAN. (2017). arXiv:1701.07875 http://arxiv.org/abs/1701.07875

[2] Edoardo Giacomello, Pier Luca Lanzi, and Daniele Loiacono. 2018. DOOM Level Generation Using Generative Adversarial Networks. *2018 IEEE Games, Entertainment, Media Conference, GEM 2018* (2018), 316–323. https://doi.org/10.1109/GEM.2018.8516539 arXiv:1804.09154

[3] Ian Goodfellow. 2016. NIPS 2016 Tutorial: Generative Adversarial Networks. (dec 2016). arXiv:1701.00160 http://arxiv.org/abs/1701.00160

[4] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. *Advances in Neural Information Processing Systems* 3, January (2014), 2672–2680. https://doi.org/10.3156/jsoft.29.5_177_2 arXiv:1406.2661

[5] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. 2017. Improved training of wasserstein GANs. *Advances in Neural Information Processing Systems* 2017-December (2017), 5768–5778. arXiv:1704.00028

[6] Pei.J Han.J, Kamber.M. 2012. *Data Mining: Concepts and Techniques.* Elsevier, Morgan Kauffman. 1–703 pages.

[7] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. 2017. GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium. Nips (2017). arXiv:1706.08500 http://arxiv.org/abs/1706.08500

[8] Antonios Liapis, Georgios N. Yannakakis, and Julian Togelius. 2013. Sentient Sketchbook : Computer-Assisted Game Level Authoring. ACM, Chania. https://www.um.edu.mt/library/oar/handle/123456789/29607

[9] Mehdi Mirza and Simon Osindero. 2014. Conditional Generative Adversarial Nets. (2014), 1–7. arXiv:1411.1784 http://arxiv.org/abs/1411.1784

[10] Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. (nov 2015). arXiv:1511.06434 http://arxiv.org/abs/1511.06434

[11] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. 2016. Improved techniques for training GANs. *Advances in Neural Information Processing Systems* (2016), 2234–2242. arXiv:1606.03498

[12] Anurag Sarkar, Zhihan Yang, and Seth Cooper. 2017. Controllable Level Blending between Games using Variational Autoencoders. (2017). arXiv:arXiv:2002.11869v1

[13] Noor Shaker, Julian Togelius, and Mark J. Nielson. 2016. *Procedural Content Generation in Games*. Springer Nature. 1–234 pages.

[14] Noor Shaker and GN Yannakakis. 2009. Towards automatic personalized content generation for platform games. *Proc. 6th Artif. Intell. Interactive Digit.* Hudlicka 2008 (2009). http://www.aaai.org/ocs/index.php/AIIDE/AIIDE10/paper/viewPDFInterstitial/2135/2546

[15] Sam Snodgrass and Santiago Onta. 2015. Controllable Procedural Content Generation via Constrained Multi-Dimensional Markov Chain Sampling. MdMC (2015), 780–786.

[16] Adam Summerville. 2018. Expanding Expressive Range : Evaluation Methodologies for Procedural Content Generation. AIIDE (2018), 116–122.

[17] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K. Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. 2017. Procedural Content Generation via Machine Learning (PCGML). (feb 2017). arXiv:1702.00539 http://arxiv.org/abs/1702.00539

[18] Julian Togelius, Renzo De Nardi, and Simon M. Lucas. 2007. Towards automatic personalised content creation for racing games. In *Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games, CIG 2007*. 252–259. https://doi.org/10.1109/CIG.2007.368106

[19] Ruben Rodriguez Torrado, Ahmed Khalifa, Michael Cerny Green, Niels Justesen, Sebastian Risi, and Julian Togelius. 2019. Bootstrapping Conditional GANs for Video Game Level Generation. (2019). arXiv:1910.01603 http://arxiv.org/abs/1910.01603

[20] Vanessa Volz, Jacob Schrum, Jialin Liu, Simon M. Lucas, Adam Smith, and Sebastian Risi. 2018. Evolving Mario Levels in the Latent Space of a Deep Convolutional Generative Adversarial Network. (may 2018). arXiv:1805.00728 http://arxiv.org/abs/1805.00728