A Generation System of 2D Video Games Levels Using TOAD-GAN

Jason Ravagli
Università degli Studi di Firenze
jason.ravagli@stud.unifi.it

Abstract

In this work we present an automatic generation system of 2D tile-based video games levels, which is composed of two parts. The first one is a Machine Learning environment based on TOAD-GAN [6] for training deep generative models with only one example level. The second part is a GUI application that allows the user to manually design levels and to generate new levels using trained models. We studied the application of TOAD-GAN to different types of levels in order to find an optimal setting for it and not to require the user to tune the network hyperparameters when applying it to new levels. The purpose of the work is to provide a ready-to-use and game independent system to design and generate game levels.

1. Introduction

Design of contents for video games (e.g. characters, levels, quests) is a process that requires a lot of time end effort, and a certain amount of creativity to produce good quality results. For these reasons, Procedural Content Generation (PCG), which is the process of automatically generate game content using algorithms, has been devised and widely used since the dawn of the video games industry.

PCG not only reduced the effort required by designers and developers to create big amounts of game content but also defined new video game genres. Indeed, the so-called Roguelike genre identifies a set of role-playing video games consisting of exploring procedurally generated dungeons. The name comes from the video game *Rogue*, a turn-based dungeon-crawling game with procedurally generated grid levels released in 1980. Actually, the need behind the PCG idea was the memory cost: with the technology of that time it was prohibitive to store huge varieties of contents, but using an algorithm that creates those contents on the fly only a little memory space was required. Nevertheless, PCG remains a common practice even nowadays.

The main limit of PCG is that the generation algorithms are game-specific and developers have to design ad-hoc solutions for their works. Hence, over the years generalized algorithms and methods for PCG have been proposed [5] [2] [1], but they remained limited to specific game genres. Recently the research has focused on the PCG via Machine Learning (PCG-ML), given the flexibility of deep learning techniques and the progress made with generative models like GANs. Various PCG-ML methods tries to exploit GAN models to automatically identify and extract patterns from existing contents and generate new ones. However, deep learning models need many examples to correctly extract and generalize patterns, and since data scarcity is the main reason to use PCG some modifications to classical GAN methods have been proposed.

Focusing on the generation of 2D tile-based levels (the ones used in platformer side-scrolling games, or in the already mentioned Roguelike games), Awiszus et al. [6] proposed TOAD-GAN, an architecture inspired by SinGAN [7] capable of generating levels with only one training example. This idea can potentially reduce to the minimum the effort required by developers to design game levels. The authors applied TOAD-GAN to Super Mario Bros. levels generation, using the publicly available collection of examples.

In this work, we want to investigate its effectiveness with general user-defined levels. Starting from the original PyTorch code, we implemented a training system for TOAD-GAN using Tensorflow. Along with this, we implemented a GUI application that allows users to design their own levels and to use trained TOAD-GANs to generate new levels. The purpose of the whole system is to provide a user-friendly game-independent tool to generate 2D tile-based levels. TOAD-GAN results are evaluated on levels belonging to two different types of games: a side-view platformer game (like Super Mario) and a top-view dungeon-crawling game. In level generation also functional requirements (e.g. playability) are important, but considering them is out of the scope of this work.

2. PCG-ML Method

As mentioned above, TOAD-GAN is based on SinGAN, a particular architecture that uses GANs to generate images using only a single example in the training process. In the following sections, we briefly describe what GANs are. We

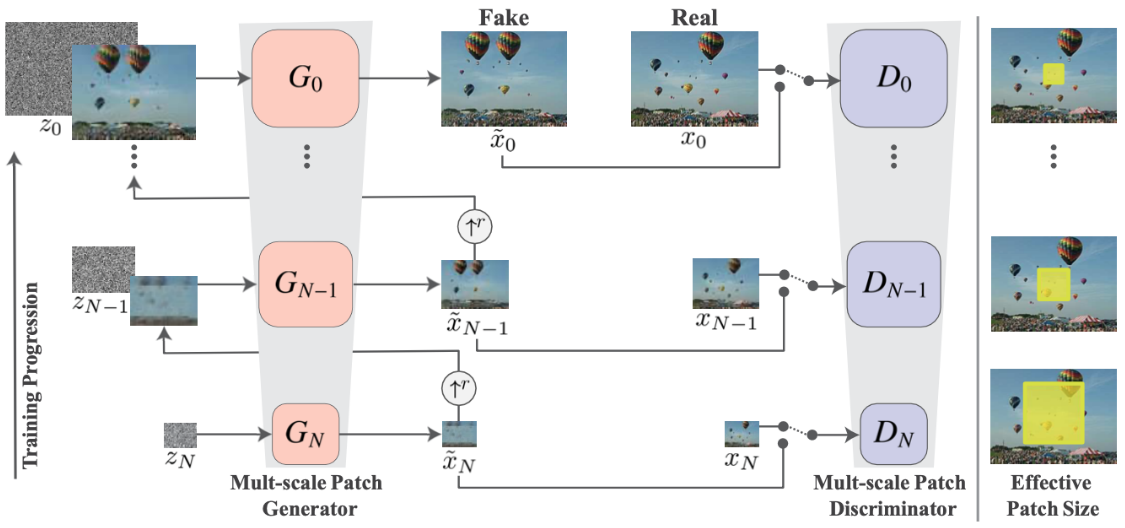


Figure 1: The SinGAN multi-scale pipeline. Each scaled version of the original image is associated with a GAN in the hierarchy. The generation process start at the coarsest scale (at the bottom), and generated images are upscaled and sent as input to upper GANs. Generators and discriminators act always on patches of the same size, focusing on finer details as we go up in the hierarchy.

then focus on the SinGAN architecture and how it works and finally we show how TOAD-GAN adapts SinGAN to PCG-ML.

2.1. Generative Adversarial Networks

Generative Adversarial Networks (GANs) were first proposed by Goodfellow et al. [3] and they are composed of a generator model and a discriminator model. The generator maps a random noise vector (called latent vector) to a generated sample, while the discriminator is a binary classifier that tries to distinguish between real and generated (fake) samples. The generator and the discriminator are trained at the same time, from which the *adversarial* name: the two networks are adversarial in a zero-sum game, where the generator adapts itself to generate samples to fool the discriminator, and the latter tries to improve its ability to distinguish between fake and real samples. GANs are fascinating networks and they have shown promising and sometimes impressive results in image generation tasks [8]. However, the training process of a GAN is unstable, the generator and the discriminator must be kept in balance during it to achieve good results, and various problems must be faced.

The WGAN-GP [4] is an extension of the original GAN that tries to improve and stabilize the training of the involved models. WGAN-GP uses a particular loss composed of a Wasserstein distance, that has better theoretical properties compared to the canonical binary cross-entropy, and a gradient penalty term for the discriminator, that prevents it from diverging producing a poorly trained generator. Since we are no more using the binary cross-entropy, in a WGAN-GP the discriminator is also called the critic. Indeed it does not calculate the probability of the input sample to be real or fake but calculates a score of its realness or fakeness. Even though they have proven to be a step forward classical GANs, WGAN-GPs remain fragile networks and continue to suffer from problems like mode collapse, where the generator produces very similar samples with few features that the critic cannot evaluate correctly. Furthermore, like other deep learning methods, they require a consistent amount of data to produce good quality models.

2.2. SinGAN

SinGAN [7] is a deep learning architecture based on WGAN-GPs that allows training a generative model of images from a single example image. Using a cascade of WGAN-GPs, this architecture acts on patches of differently scaled versions of the training example. An intuitive scheme of SinGAN is shown in Figure 1. Each GAN in the hierarchy is associated with a scaled version of the training image. The generator is responsible for mapping a 2D gaussian noise tensor to an image having patches from the same distribution of the assigned scaled image.

Let us examine deeper how SinGAN works. The generation process follows a bottom-up approach, starting from the coarsest scale (the lowest) of the hierarchy and sequentially passing through all the generators up to the finest scale. At the coarsest scale, the generator simply maps noise to an image. The other generators instead have to add details to the image generated at the previous scale. For this reason, besides a noise tensor, they take as input also the upsampled image provided by the previous scale generator.

Figure 2 shows the functioning of a single generator.

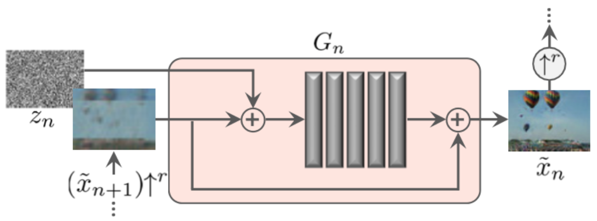


Figure 2: Functioning of a generator at scale n. It receives as input a noise tensor and the upscaled version of the image generated at the previous scale (n+1). Noise and input image are added together and sent to a network with 5 convolutional blocks. The output of the network is then added again to the input image to produce the final generated image

Noise and upsampled image are added together and sent to a convolutional network. Appropriate padding is added to have the output image with the same size as the input. Finally, the upsampled image from the previous scale is added to the output. All generators and critics have the same receptive field, hence they focus on finer details and patterns as we go up in the hierarchy. The generator and the critic are composed of 5 convolutional blocks, each formed by a convolutional layer with 3x3 filters, a batch normalization layer and Leaky ReLU as the activation. The models are fully convolutional and independent of the particular image size: the output image dimensions are given only from the size of the input noise matrix. This allows a trained SinGAN to generate images of different sizes having patches from the same distribution as the original one.

A SinGAN is trained sequentially, from the lowest GAN in the hierarchy up to the highest one. Once a GAN is trained it is kept fixed and we start training the next one. While for the critic the canonical WGAN-GP loss is used, in the training of the generators it is used the following one:

$$L_{adv} + \alpha * L_{rec}$$

The first term is called adversarial loss and corresponds to the generator loss of a WGAN-GP. The second term is called reconstruction loss and it is multiplied by a constant weight factor α with a typical value of 0.1. The reconstruction loss constrains the latent spaces to contain a specific set of noise tensors that generates the original training image. These noise tensors (called reconstruction noise tensors) are chosen before training and they are all zero except at the coarsest scale, where it is equal to some fixed z^*. At each training step, the SinGAN is fed with the reconstruction noise tensors and the generated image is compared with the training image at the current scale. The reconstruction loss is then calculated as the sum of pixel by pixel squared differences between the two images. Let x_n be the scaled training image at the n-th scale and N the number of scales

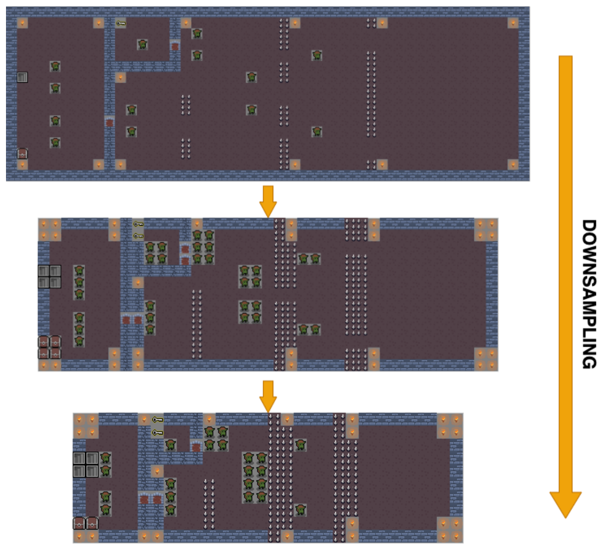


Figure 3: The downsampling procedure applied to an example level. More important tiles, like the enemies, chests an doors, are kept during the scaling operation at the expense of less important tiles like ground and walls.

in the hierarchy. For $n < N$ (i.e. not at the coarsest scale) the reconstruction loss is calculated as

$$L_{rec} = \left\|G_n(0, (x_{n+1}^{rec}) \uparrow) - x_n\right\|^2$$

where G_n is the generator at scale n and $(x_{n+1}^{rec}) \uparrow$ is the upsampled reconstructed image from the previous scale. For $n = N$ instead, the reconstruction loss becomes

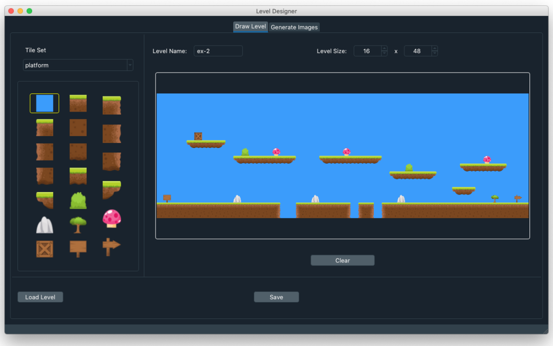
$$L_{rec} = \|G_n(z^*) - x_N\|^2$$

The reconstructed image at scale $n + 1$ is also used to determine the standard deviation σ_n of the noise generated at scale n, which controls the amount of detail that needs to be added at that scale. σ_n is taken to be proportional to the root mean squared error between $(x_{n+1}^{rec}) \uparrow$ and x_n by a factor of γ.

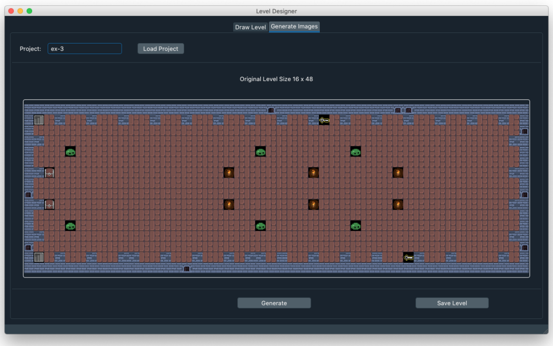
2.3. TOAD-GAN

SinGAN was conceived for natural images. Awiszus et al. with their TOAD-GAN (Token-based Oneshot Arbitrary Dimension GAN) [6] adapted SinGAN to the generation of 2D tile-based levels.

Each tile in a level is considered as a one-hot encoded vector (a token) and can be seen as a pixel in an image. The length of the one-hot encoding is determined by the number of different tiles in the training example level. Since each token corresponds to a pixel, simply downsampling to calculate the scaled images for training would result in a loss of information. Indeed, aliasing would make important tokens



(a) Manual design of levels



(b) Automatic level generation using trained TOAD-GANs

Figure 4: Screens of the GUI application

disappear. Hence the authors proposed a downsampling procedure that considers the importance of tokens. Tokens are organized in a hierarchy built a priori: tokens that are lower in the hierarchy are more important and in the downsampling procedure will be retained at the expense of adjacent less important tokens. Having a collection of different training levels, the importance of a token can be calculated as the frequency of a token multiplied by the inverse of its level frequency (i.e. the number of levels in which the token appears). A token that appears often and in many levels is obviously less important than rare tokens that appear only in specific types of levels. With the value of importance of each token we can build the aforementioned hierarchy. Figure 3 shows the downsampling procedure applied to an example level.

TOAD-GAN appears to be a promising tool in the PCG-ML field.

3. Implemented System

Our system is composed of a Tensorflow environment to train a TOAD-GAN on a custom level and a GUI application that allows the user to draw the levels that will be used for training the networks and to generate new levels using trained TOAD-GANs.

3.1. Training Environment

Starting from the original PyTorch project, we implemented TOAD-GAN and its training procedure using Tensorflow. Since TOAD-GAN is not an ordinary deep learning model and needs additional metadata in order to functioning (e.g. the GAN hierarchy, the noise standard deviation and the reconstruction noise at each scale), to easily save and reload the network we defined the TOAD-GAN project. A TOAD-GAN project stores and organizes all the information required to build and use the network, along with useful information about its training process, such as its training image, loss plots and reconstructed images at each scale. We also rationalized the concept of token hierarchy in a file format as game-independent and easy as possible, to allow users to define their own token sets and hierarchies. The training environment can be use through a command-line interface, it takes a training level and the related token hierarchy and produce a TOAD-GAN project as output.

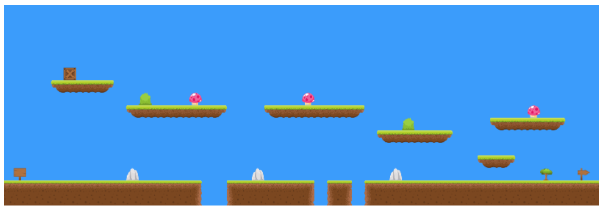
3.2. GUI Application

The GUI application has been implemented using Python and PyQt5. Figure 4 shows the two main screens of the application. The user can design levels (Figure 4a) by selecting a tile from the toolbox to the left and then clicking and dragging the pointer on the drawing area. The tileset is chosen through a drop-down menu. We provided two default tilesets (the same that we used in our experiments), namely a platform tileset and a dungeon exploration tileset, but users can import custom tilesets by creating a file following the format used for token hierarchies and putting it in the application folders, along with the tiles images. Created levels can also be saved and reloaded for further editing. The application can load TOAD-GAN projects and use them to generate new levels (Figure 4b). Generated levels can be saved to files.

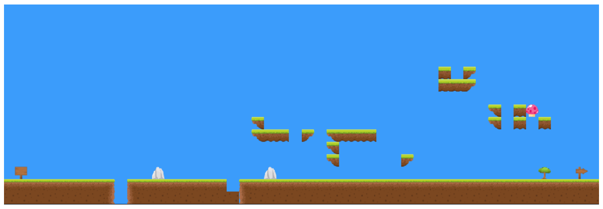
4. Experiments and Results

The original paper presented TOAD-GAN applied to the generation of Super Mario Bros levels. They have the same peculiarities as most 2D sliding platformer games: they are very extensive in width but they have a small height, with uniform patterns at the top and bottom forming the ceiling and the floor. In our experiments, we want to test TOAD-GAN on different types of levels, both in size and style.

We considered two different tilesets to design our test levels: the first is a simple 2D platformer tileset, just like the Super Mario one, while the second is a top view fantasy-



(a) Original



(b) Generated

Figure 5: TOAD-GAN with default settings applied to a level with floating platforms. The network struggles to organize central tiles coherently.

style dungeon tileset.

4.1. Exploratory Tests

We first performed some exploratory tests using TOAD-GAN with the default settings from the original implementation to analyze its behavior. We found that the network can identify and accurately replicate patterns at the edge of the level, like the floor and the ceiling in a platformer game, as long as they are quite simple. More but not too complex patterns can be learned by the network if they are sufficiently repeated along a level axis. The situation changes when we analyze the central parts of the generated levels: in general, a game level presents small located patterns in the central area, and TOAD-GAN struggles to learn them. Generated levels tend to have central parts with partial patterns and random isolated tiles, perhaps caused by noise. In levels like those with floating platforms, where tiles must be organized coherently, the network gives poor results (Figure 5).

4.2. Hyperparameters Tuning

Having an idea of which are the strengths and weaknesses of TOAD-GAN, we moved forward and tried to find a setting for the hyperparameters that could make the network perform well on as many types of levels as possible. Doing this, along with the GUI application, we want to provide the user with a ready-to-use tool that requires little to no knowledge about its implementation details. The considered hyperparameters are 5:

- the number of epochs
- the factor γ for the calculation of the noise standard deviation

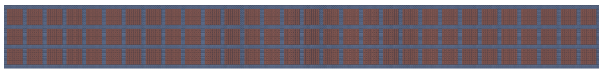


Figure 6: The simple and repetitive pattern used to tune the hyperparameters

- the number of image scales and GANs in the TOAD-GAN hierarchy
- the patch evaluation method used by the critic
- the number of convolutional blocks used in generators and critics

Experiments were conducted on a special level we designed, shown in Figure 6. It is composed of a very simple square pattern (a room surrounded by walls) repeated both in height and width. This level is considerably wide but not so tall in order to analyze the capability of the network to learn the same pattern in the same example but with different extensions.

When working with WGAN-GPs, in general we cannot have a quantitative measure of how well the network is performing. Especially with TOAD-GAN, the only things we can do to check if the training process went well is to observe the generator and critic losses ad each scale (they have to be as closer as possible and not to diverge) and to ensure that the final reconstructed level matches the training example. For these reasons, to evaluate the quality of the generated levels we defined a set of qualitative criteria, specifically devised for our synthetic training example:

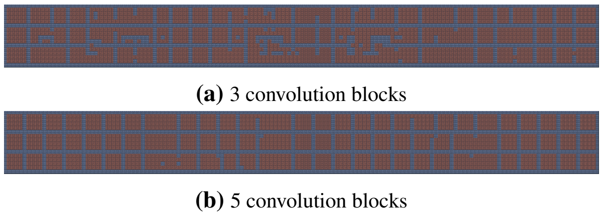
- edge patterns replication
- horizontal patterns replication (the two rows of walls that divide the level into three parts)
- vertical patterns replication (the columns of walls)
- square patterns replication and randomization (the rooms)
- level cleanliness

Starting from the default values provided by the original implementation, we studied the effects of each hyperparameter on the generated levels and tried to propose an optimal setting. Table 1 summarizes the results. Let us analyze them singularly.

The default number of training iterations is 4000 for each GAN in the TOAD-GAN hierarchy. In principle, a WGAN-GP can be trained for a high number of iterations with no drawbacks, as long as the training is stable and generator and critic losses are in balance. Indeed TOAD-GANs trained for more epochs learned better all patterns and replicated them in a cleaner way. 8000 iterations was a good

Hyperparameter	Default	Optimal
Epochs	4000	8000
γ	0.1	0.05
N Scales	4	$3 \sim 4$
Critic Evaluation	mean reduction	mean reduction
N Conv. Blocks	3	5

Table 1: Comparison between the default values of the hyperparameters provided by the original implementation and the optimal ones found


(a) 3 convolution blocks

(b) 5 convolution blocks

Figure 7: Levels generated by TOAD-GANs with 3 and 5 convolutional blocks trained on the pattern example

tradeoff between training time and network quality. Since the learning rate is decreased by a factor of 10 after a fixed number of iterations, these thresholds had to be changed accordingly.

The γ factor determines the standard deviation of the input noise, and consequently its amplitude. A value of 0.1 generated levels with frequent interruptions and discontinuities in wall patterns, and dirty levels with random isolated tiles, apparently caused by noise. On the contrary, setting alpha to 0.01 made the network less capable of reconstructing the original image. We found that 0.05 was a good choice.

The original implementation builds TOAD-GAN using a hierarchy of 4 GANs, trained with the original image scaled by a list of fixed factors. However, we found it more appropriate to adjust the number of scales according to the height of the training level. More specifically, we considered the receptive field of the last convolutional layer of our GANs: we used a number of scales so that this receptive field in the last scale is greater than or equal to the image height. This way the last scale is capable of analyzing global patterns of the level. The receptive field can be easily calculated as the number of convolutional layers plus one, hence the only parameter that determines the depth of the hierarchy is a single scaling factor that we repeatedly apply to the training image. We observed that with a too deep hierarchy, going up in it during the training process the critics and generators losses were more and more separated. The entire training remained stable, but the final TOAD-GAN was not able to reconstruct accurately the example level, and tiles in generated levels were placed almost randomly, not forming well identifiable patterns. Good results were achieved using 3 or 4 scales. In conclusion, the scaling factor has to be chosen considering this target number of scales, the number of convolutional layers in the networks and the height of the training example.

As stated in the previous sections, the critics used by TOAD-GANs are fully convolutional, and substantially they assign a score of realness to each patch extracted from the input level. In the original implementation, these scores are averaged to calculate the whole level score for the training loss. We tried to use a more severe loss for the critics, calculating the whole level score as the maximum patch score of realness for the fake images and the minimum score of patch realness for the real images. This way the critics would have judged a real level as fake if there is at least one patch recognized as fake, and a fake level as real if there is at least one patch recognized as real. However, the resulting networks were not able to reconstruct the example level and to learn patterns correctly. The generated levels also presented random clusters of tiles.

As the last hyperparameter, let us talk about the number of convolutional blocks in generators and critics. SinGAN used 5 convolutional blocks while TOAD-GAN used only 3 blocks as the default value. We investigated how this choice affects the networks and results. First of all, as explained above, the number of convolutional blocks determines the receptive field of the convolutional layers and consequently the number of scales in the network: with the same scaling factor fewer blocks means more scales required. The convolutional blocks have also a huge impact on patterns learned. Using 5 blocks the replicated patterns were neat and clean, suggesting that could be the best choice. However further tests on different levels showed that with 5 blocks the generator suffered from partial mode collapse: generated levels were very similar to the training one, with little variability. Perhaps this is caused by the models capacity, which allows them to memorize the patterns of the example. Keeping 3 blocks as in the original implementation the networks replicated and reorganized patterns in different ways, but with inaccuracies especially on vertical ones. Furthermore, levels were noisier and dirtier. Using 4 convolutional blocks not always led to clear improvements over 3-blocks and 5-blocks networks. Definitely, the choice depends on the tradeoff between variability and cleanliness that we want in our levels: 5 blocks could be preferable, but if we observe mode collapse symptoms on trained level we can lower them to 3 or 4. A comparison on the synthetic level is shown in Figure 7.

With the optimal set of hyperparameters that we found we applied TOAD-GAN to different levels designed using the two types of tilesets presented above (the platformer one and the fantasy-style dungeon one). We experimented with different level shape, including a square level. We observed that with big levels, extended in both height and width, the network suffered from mode collapse, especially using 5 convolutional blocks. Some results of the experiments can be found in the appendix.

5. Conclusions and Future Works

The implemented system can be a useful and convenient tool for designing 2D game levels through PCG. TOAD-GAN is a promising network in the PCG-ML field, but it is very fragile and presents some limitations. Even when using an optimal setting, with some particular training examples it generates too noisy or randomized levels and requires being careful when adjusting hyperparameters. Furthermore, it does not consider functional constraints in levels, so it is required a companion tool that verifies, for example, the navigability of levels.

It would be interesting to extend the TOAD-GAN idea to handle multiple training examples. The training examples should be variant of the same levels, to effectively train the network and to prevent it from generating ”chimera” levels. Perhaps this would avoid the mode collapse problem when using more capable GANs keeping the good properties of cleanliness and pattern replication.

References

[1] Steve Dahlskog and Julian Togelius. Patterns and procedural content generation, 2012. 1

[2] Lucas Ferreira and Claudio Toledo. Generating levels for physics-based puzzle games with estimation of distribution algorithms, 2014. 1

[3] Mehdi Mirza Bing Xu David Warde-Farley Sherjil Ozair Aaron Courville Ian Goodfellow, Jean Pouget-Abadie and Yoshua Bengio. Generative adversarial nets, 2014. 2

[4] Mart´ın Arjovsky Vincent Dumoulin Ishaan Gulrajani, Faruk Ahmed and Aaron C. Courville. Improved training of wasserstein gans, 2017. 2

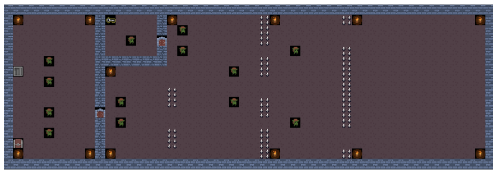
[5] Leonardo Pereira Lucas Ferreira and Claudio Toledo. A multi-population genetic algorithm for procedural generation of levels for platform games, 2014. 1

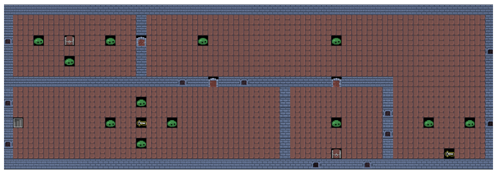
[6] Frederik Schubert Maren Awiszus and Bodo Rosenhahn. Toad-gan: Coherent style level generation from a single example, 2020. 1, 3

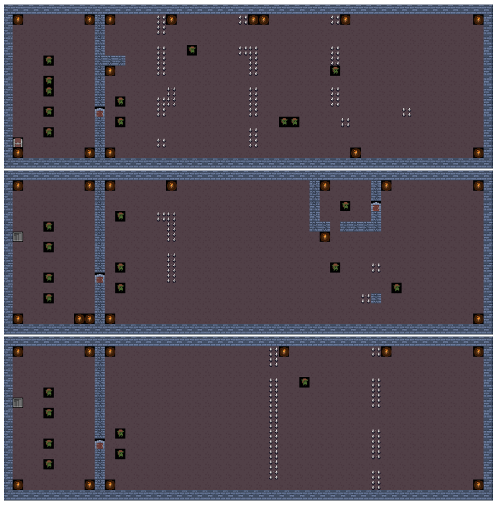
[7] Tomer Michaeli Tamar Rott Shaham, Tali Dekel. Singan: Learning a generative model from a single natural image, 2019. 1, 2

[8] Samuli Laine Tero Karras and Timo Aila. A style-based generator architecture for generative adversarial networks, 2019. 2

A. Examples of Application


(a) Original


(b) Original


(c) Generated

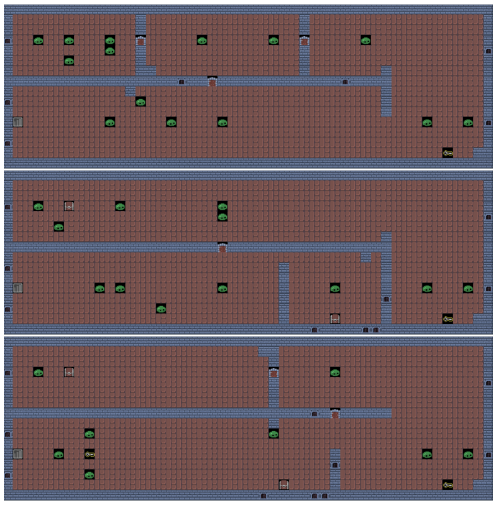
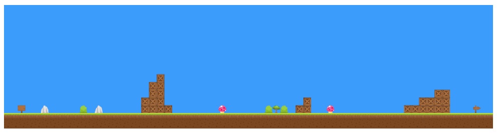
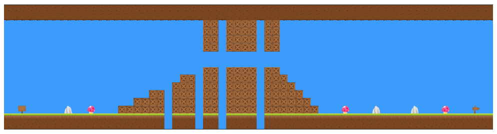
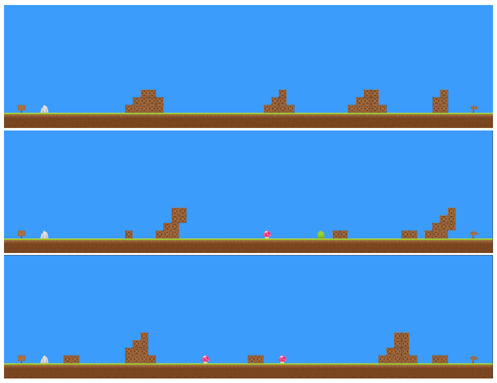

(d) Generated

Figure 8: TOAD-GAN with optimal hyperparameters and 5 convolutional blocks applied to fantasy-style dungeon levels


(a) Original


(b) Original


(c) Generated

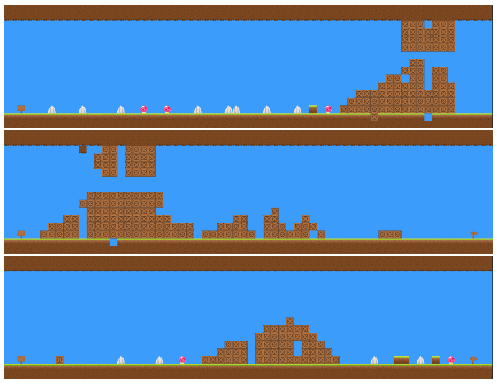
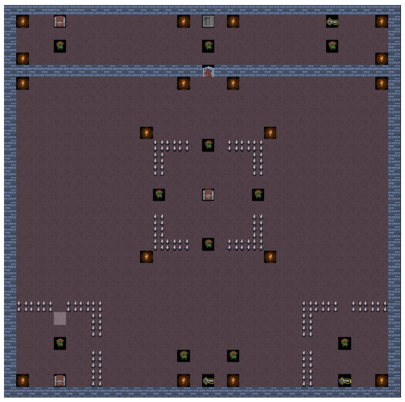
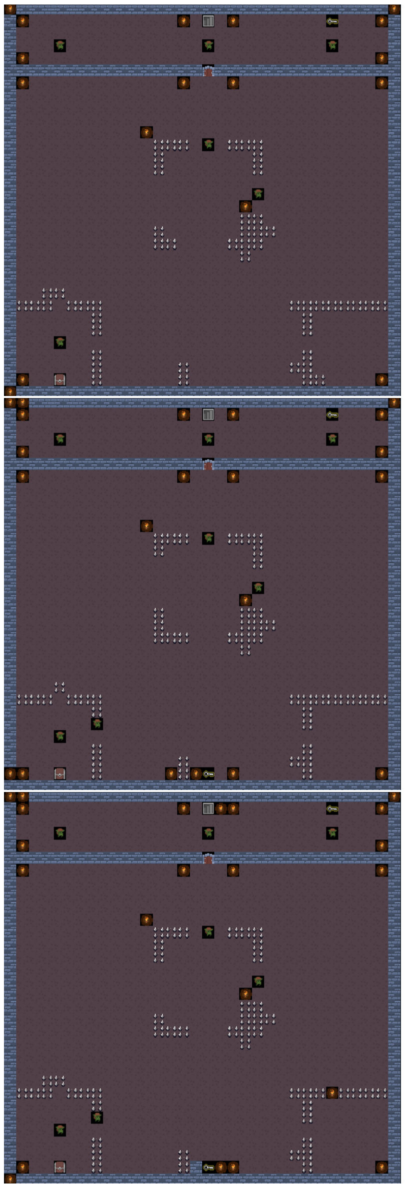

(d) Generated

Figure 9: TOAD-GAN with optimal hyperparameters and 5 convolutional blocks applied to platformer levels

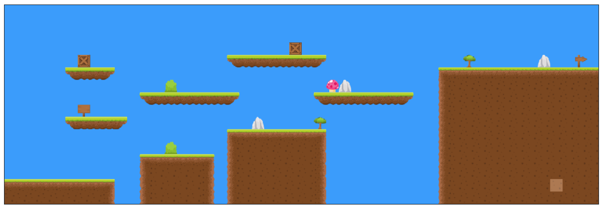


(a) Original

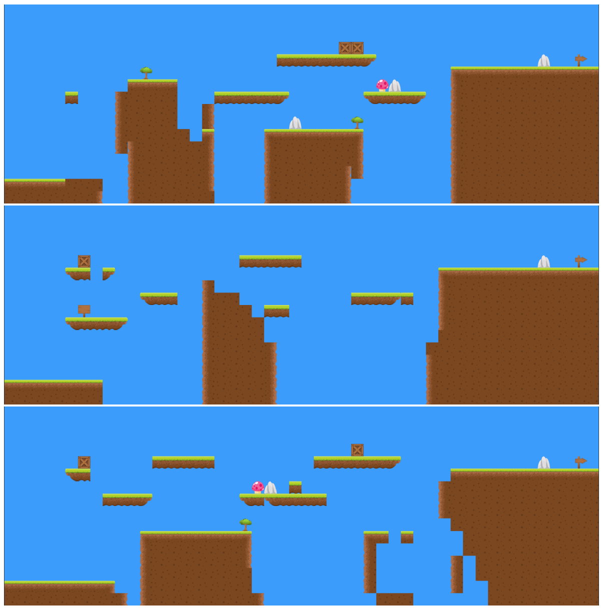


(b) Generated

Figure 10: When applied to big levels extended both in height and width, TOAD-GAN suffers from mode collapse



(a) Original



(b) Generated

Figure 11: When using levels with central patterns that requires a certain functional organization or levels with edge patterns that extend also in the central areas, TOAD-GAN struggles to generate good quality results