Conditional Convolutional Generative Adversarial Networks Based Interactive Procedural Game Map Generation

Kuang Ping and Luo Dingli(✉)

University of Electronic Science and Technology of China,
Chenghua 610051, Sichuan, People's Republic of China
kuangping@uestc.edu.cn, godofpen05@gmail.com

Abstract. There is a strong need for a procedural map design system which can generate complex detail game maps but with simple user control. We propose an interactive real-time design system made with Conditional Generative Adversarial Network and Convolutional Neural Network. This system takes user-defined game-play area map as input, and generate a complex game map with the same design pattern as training samples automatically. It can output an abstract label map which can be used in other procedural generator called theme renderer. The impacts of our obtained results show the potential of deep learning methods used in procedural game map generation.

Keywords: Generative adversarial networks · Procedural generation · Convolutional neural networks

1 Introduction

It is necessary to design a great game map when you develop a great game. As the size of the game world is getting bigger and bigger, the dimension of game maps becomes larger and larger. Considering the huge cost of manually design the big game world, developers are more interested in programming game maps with the help of algorithms. However, the design of game map needs human ideas, which means it cannot be replaced easily by simple algorithms. It needs to consider the game-play design, the sense of vision and the style of the whole game. These are not easy to express manually by rules or codes. Therefore, a more reasonable approach is to use neural networks to learn these rules from a set of examples and to control the basement shape through Designers' interactive tools. Finally, the network generates detailed results depending on both the map style from examples and the shape from designers.

There are many methods for procedural generation, which can be divided into three kinds: rule-based generation, noise-based generation and example-based

This work supported by Sichuan Science and Technology Program 2019ZDZX0009, 2019ZDZX0005, 2019GFW1116 and 2018GZ0008.

K. Arai et al. (Eds.): FICC 2020, AISC 1129, pp. 400–419, 2020.
https://doi.org/10.1007/978-3-030-39445-5_30

generation. The first kind of methods usually use a set of manually designed generation rules like 'generate a **door** at a **wall** between two **rooms**'. Take dungeon generation as an example, the dungeon consists of a different set of different usage rooms, each room has regular and straightforward shapes. This makes rules can describe the dungeons. Rules-based approaches have been proved to be useful in dungeon generation. Noise-based methods are more popular for generating outdoor worlds, such as giant planets with complex terrain features. These methods can generate non-redundant scenery for the result, which is more suitable for terrain generation than rule-based method. The example-based approach has been used over the years, and its basic idea is to learn rules from a series of examples and produce new results. With the improvement of machine learning and deep learning, many developers use neural network as a powerful tool based on example method. Generating antagonistic network (GAN) is a network that can generate results based on a set of given examples without giving any rules manually, which is a powerful tool to accomplish this task.

The combination of these three approaches is a more useful solution for industrial use. We introduce a solution including cGAN (extension of GAN), convolutional neural network (CNN) and traditional procedural methods as post-processing. In order to learn the design style of a given example, the neural network is trained by a large number of designed game maps. Also a conditional mask drawn by the designer will be used as the input of the cGAN to control the generation. As a result, the solution takes into account the randomization of the generated results, the play-ability area style of the game and the control of the designer.

In order to achieve our solution, we need a designer's basic mask. cGAN will use masks as input to generate a generated label map, each label is represented as a map element. Label maps will be processed by other layers of processors, such as CNN, which is used to connect each game area processor, or CNN, which is based on traditional processors used to generate terrain heights. Finally, the theme rendering program will be used to generate the final 2D game map or even 3D game map. All the above steps can be completed in 1–2 s. This solution enables designers to change results interactively by editing a part of the input mask map.

The main contributions are as follows:

1. We propose an interactive procedural game map generation solution based on Deep Learning. This solution can automatically divide the rooms and areas, and generate terrain and decorators based on a mask map.
2. We propose a group of neural networks contains cGAN and CNN in order to generate a label based game map.
3. We show a set of different testing in order to prove our method can be used as many different kinds of game map generation.

2 Related Works

Procedural content generation has been discussed in many papers. Article [6] contains an overview about different kind of procedural generation methods. We

just make a summary of these methods by dividing them into three kinds: rule-based generation, noise-based generation and example-based generation. This part also introduces the related information of deep learning and neural network.

2.1 Rule Based Generation

Rule-based generation is based on the idea that the generation schedule can be described by a set of relevant rules, most of which are described as grammar trees. Each rule contains some operation and a set of sub-rules. By executing the rules from root to leaf, it can generate objects or modifies the generated results. This method can be used to generate game maps, dungeons, etc. Article [8] and [24] are examples of projects generated for role-playing games. For industry purposes, many games are powered by these methods such as Diablo and Darkest Dungeon. These methods usually require designers to build a lot of small parts of the game map and combine them according to manually defined rules. Also, Diablo also uses this method to generate names for inventory. These names are divided into prefixes, types and other parts, and then, use rules to generate names and the ultimate ownership of inventory, so that the inventory volume corresponds to its name. In another area, the L-system is a powerful tool for generating trees and plants.

2.2 Noise Based Generation

In order to generate randomized results, noise is a good choice. Perlin noise; for example, has been widely used in different procedural generation methods. Author in [25] shows a method for procedurally generating clouds with Perlin noise. Researchers have introduced many terrain generation solutions based on different kinds of noise combinations; for example, [28] are using noise functions to generate terrain with complex features. These methods tend to use noise functions as following steps: using a noise function as an original height map, which defines the height at each position of terrain, and then using many noise functions to modify the result in order to simulate the erosion. By computing on GPU, terrain can be generated in real time [23]. However, it is difficult to control the generation results directly, even if we can easily control the mixture method or mixture weight of different noise function. Therefore, even though the noise-based generation can generate good random results for terrain and cloud, it is still difficult to directly control the result generation based on noise.

2.3 Example Based Generation

Because designers want better control over the results, and they do not want to spend too much time on details, a reasonable approach is asking the user to make sketches to show the basic idea of their design and use methods to adding details from existing examples. This method is also widely used in terrain generation; for example, [30] considers this problem to be a patch matching problem. By

matching the height field from real terrain into user-defined sketch, a new terrain contains complex detail has been produced. Methods allowing users to design the result interactively are called interactive terrain editing. Papers like [9,10] introduce good solutions in this kind of methods.

Traditional patch fitting methods may produce correct results in some cases, yet deep learning approaches can provide better results in example-based generation area; for example, [13] provides an interactive terrain editing system based on the conditional generative adversarial network. The network in this paper is trained based on a set of real terrain information and predicts the results in real time according to the designer's sketch.

2.4 Deep Learning in Procedural Generation

By the improvement of machine learning, deep learning approaches are widely used for generation. A really powerful network is Generative Adversarial Network [12]. By training two adversarial networks: generator and discriminator, the generator will get the ability for generating same style results from given examples. Since the discriminator is training during this process, it provides a gradient for optimizing generator. However, original GAN only takes a random input vector Z as the generator's input, which means the user can not control the results. Conditional generative adversarial network [22], an improvement of GAN, allows a user control by concatenating the user control input vector Y with original input Z as a new input vector. To use an image as input, a popular solution is using a U-Net [27] network architecture. This network is based on Convolutional Neural Network [19], which take one or more images as input, and processed by many convolution layers, then provide a final result. A combination of U-Net network and conditional GAN is Pix2Pix [16]. It can be used to take one image as input and generate another image based on the input. This feature makes this network suitable for procedural terrain generation, like in [13].

3 Overview

As shown in Fig. 1, our pipeline contains three different steps. The user's input is a mask map, where the white area represents the shape of the game area. This map will be given to conditional GAN(cGAN) as input. Then, cGAN will generate an image with many channels(usually more than one channel), which we call $LabelMap$. Each channel represents a kind of game-play label. Game-play labels are an abstract representation of game-play element like the ground, mountain, tree and so on. We call cGAN generated game-play label map as $InitialLabelMap$. Then one or more convolution neural network(CNN) will use some label channels to process additional a label channel or modify a label channel. These CNN networks are called post-process CNN. Finally, a program called $ThemeRenderer$ is used to convert abstract game-play label into real game elements in the game. Like we can calculate the height of mountains based on mountain label channel. Alternatively, we can produce random size trees on

the tree label position. This allows us to generate 3D scenes based on the results although the networks learns from examples in 2D games.

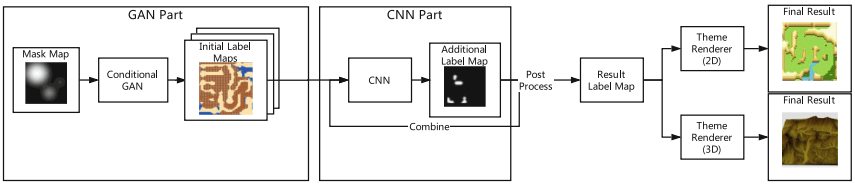

Fig. 1. Our system consists of three main steps. Designer draws a mask as input to conditional GAN, and then it generates an original label map with multi-channels. Each channel represents a label, such as water, ground or mountains. Then, the original label map will be processed by one or more CNN networks to modify the map or add additional maps. Finally, the label map will be rendered by the theme renderer, and different theme renderer will produce different results, such as 2D maps and 3D terrain.

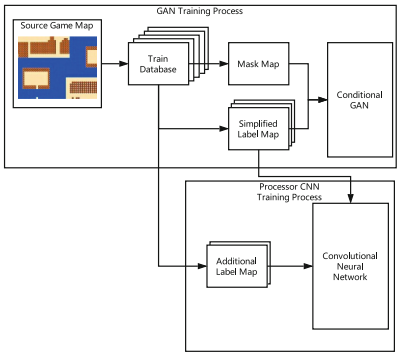

Fig. 2. The source game map is converted into a training database map, each channel containing a label data. Then it is simplified to a lower channel number map and a mask map for training conditional GAN(cGAN). Then, another label channel map is used as input to train a processor CNN network with a simplified label map.

Before use to produce results, we need to train these neural networks. The training steps are shown in Fig. 2. We use game maps from games on Nintendo Entertainment System (NES). NES game maps are usually constructed by tiles, each tile usually represents one kind of game element, which makes it easy to convert into a label map. We simplify the converted label map and then calculate the game-play area mask from labels. For example, take Ground Label Channel used as the game-play area. Then we use the game-play area mask as input and the corresponding label map as output to train the cGAN. For training CNN, it

depends on the usage of this network, and details of the training process will be explained in the following sections.

4 Neural Network Architecture

We introduce two types of neural network architecture to generate label maps. In this section, we will discuss the structure of the network and the reasons for our selection.

4.1 Conditional Generative Adversarial Network

We use a network based on conditional GAN [22], which is based on the original Generative Adversarial Network [12] and the U-Net encoder-decoder framework [16,27]. Like we shows in Fig. 3, our cGAN contains two deep networks: U-Net encoder-decoder network generator G and Convolution Neural Network discriminator D. For each training, there will be a pair of images obtained from training database: Y is a mask image convert from the gamep-lay area, X is a simplified label image. The goal is to train G with real image pair (Y, X), and then it can use the input mask image Y to generate the output label map image $G(Y)$. The discriminator network D is trained to identify the given image pair (Y, X) and $(Y, G(Y))$ and give the result about which it is. This process can be described as optimizing both two network parameters to play this min-max game:

$$IoU = \frac{\sum_{ijk} \left[I\left(y'_{ijk} > p\right) * I(y_{ijk}) \right]}{\sum_{ijk} \left[I\left(I\left(y'_{ijk} > p\right) + I(y_{ijk})\right) \right]} \tag{1}$$

In fact, in the original conditional GAN paper, it not only needs to input Y but also Z, which makes the formula contains conditional parameters. However, according to the suggestion from [13], we do not use the noise vector z and use dropout to provide randomize.

Author in [16] introduce a method to improve the training: slice the image pair (Y, X) and $(Y, G(Y))$ into a set of sub-patches with smaller size like 32×32, then use discriminator to discriminate each patch and combine them by this formula:

$$\sum_{(Y_n, X_n) \in (Y, X)} D(Y_n, X_n) + \sum_{(Y_n) \in Y} D(Y_n, G(Y_n)) \tag{2}$$

We also follow this advise and use a small number of patches since the size of our label map is small but channels are more than regular images.

We use a combination of mean squared error and categorical cross-entropy as loss function to train generator, and use binary cross-entropy as loss function to train discriminator. Since MSE measures the difference between the original map and generated results, it makes the generator provide images that close to a given mask. Our masks have the same shapes because they are converted from original label maps. Also, cross entropy measures the style difference which is

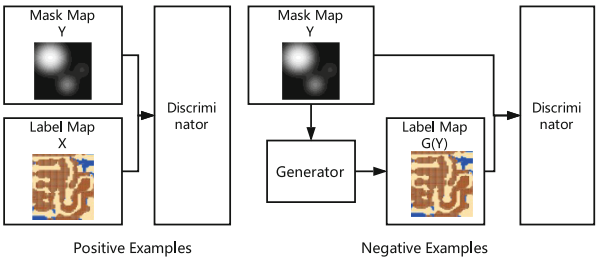

Fig. 3. Conditional Training overview: generator G takes Mask Map Y to produce $G(Y)$, discriminator D takes a pair of (Y, X) or $(Y, G(Y))$ and try to discriminate which it is.

discriminated by discriminator D. And We use RMSProp as optimize function. During our test, it provides better results than Adam, but the difference between them is not statistically significant.

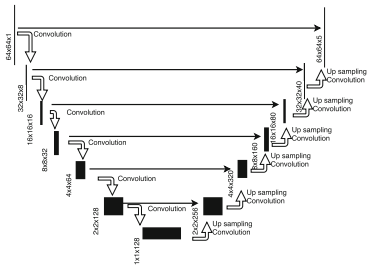

Fig. 4. The basic shape of our U-Net convolutional neural network. Compared with the original Pix2Pix paper, our layer is smaller.

4.2 U-Net Based Convolutional Neural Network

Since the generator network G and the processor CNN network shares the basic U-Net network architecture. The basic structure is in Fig. 4. we introduce them both in this part.

Basically, this network is an encoder-decoder network. However, as described in [22] and [16], U-Net concatenate the correspond size layer with lower layer's output as input for up convolution layer. This approach keeps the detail information of higher positions to avoid losing position information during encode and decode operations.

In our case, we adjusted the network from [16], but we did not use a large and deep U-Net structure because our label map's size was small (64×64), although it contained more channels. Pix2Pix architecture outputs a pixel-based image, which means that small differences do not really affect the results. However, our

approach is to category labels, which means that small differences can have an better impact. For example, changing the red channel of a pixel by 0.1 does not really change the final result, but it may lead to a larger label channel value, which will provide different elements, such as changes from tree to water. Therefore, we intend to use a simpler network architecture to make the results more stable and accurate.

Figure 7 shows an example of CNN called *Connection CNN*. It shows the basic training and usage of this network. Details are explained in the next section.

4.3 Other Networks

We also build and test an another different network as a generator called Deep Convolutional Generative Adversarial Network (DCGAN) [26]. In some cases, people do not or can not control the generate result manually, but want to generate result from a random vector Z directly. For example, in some games, there are dungeons or areas generated each time when player comes in.

We use a network adjusted from DCGAN, and followed advice from Wasserstein GAN [7], and provide more stable training progress and get usable results successfully. In this network, the generator is a decoder with a randomize input vector Z and provide an output label map the same as generator G in cGAN. We limit the weight of discriminator in $[-0.01, 0.01]$ after each training, and then train D 5 times and G 1 time in each epoch. Finally, we use softmax function as the activation function, categorical cross-entropy as the loss function, and RMSprop as an optimizer.

This network can be used to replace the generator G in our pipeline, then it can be used as a random map generator. There is no need to change the rest of our pipeline.

5 Training

Normally our network's training database contains a set of image pairs (Y, X). Y is mask image and X is a label map. However, it is not easy to draw label maps directly manually, and it is not affordable for drawing a large number of maps by professional artists. So we choose to use game maps from NES games, such as Legend of Zelda. These old game maps consist of small tiles. Each tile looks like a small image representing game elements, or in our case, a label. But there are still many kinds of tiles, we need to simplify the original label database to produce more stable training. In this section, we use maps from Legends of Zelda to show the basic training progress of our networks.

5.1 Training Database Preparation

As shown in Fig. 5, the construction of training database is completed before start training the network. Training Database is built in the following steps:

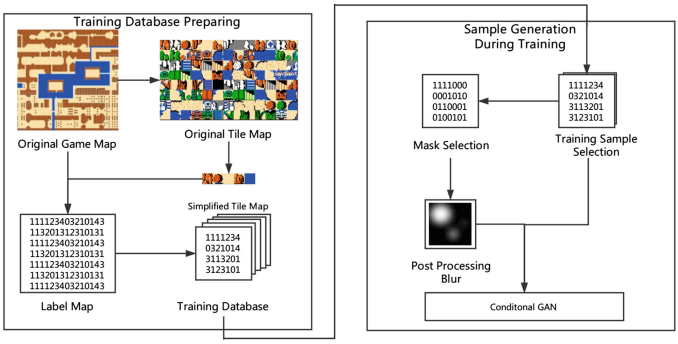

Fig. 5. This image shows the basic sample processing for training database construction and conditional GAN training. Main database processing is divided into two parts. The main database processing is divided into two parts. The first part is from the original game map to a set of label maps as training database, which is completed before training. The second part is the training input and goal from the selected samples to the conditional GAN in each epoch, which is done in each epoch.

- **Labeling.** Building a directory map containing each kind of 16×16 tiles. Each 16×16-pixel region is then marked with a directory index, which converts the result into a single-channel integer map.
- **Simplify.** Usually we have a large number of types for tiles, and we want to simplify them. In our case, we only use five kinds of tile types: Ground, Tree, Water, Mountain, Rock.
- **Slicing.** After conversion, we will have a basic label map. We want to slice the result into small parts so that they can be added to the training database. We converted the map into 64×64 patches, 32×32 patches, 16×16 patches. Patches smaller than 64×64 will be scaled into 64×64.
- **Rotate and Flip.** In order to add more training samples, we rotate and flip each patch.
- **Sample filtering.** We use a filter to separate some unusable patches, which do not contain any game-play areas.

Finally, we obtain a single-channel training database, which will be extended to a five-channel training database at run time.

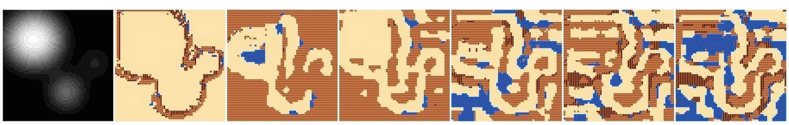

Fig. 6. This image shows the same mask (first image) as input and the results in training progress. Epoch number from left to right is 10, 100, 190, 290, 390, 550. It shows that the network can generate small rooms if the mask area is large.

5.2 Conditional GAN Training

For training the conditional GAN, we choose to generate the training samples on every epoch instead of before the training begins. We have tested preparing all of the training samples at the beginning. It takes up huge memory. So we decided to prepare this single channel 64×64 image as a sample database for middle results. In each training epoch, we only convert the images selected by this epoch into the two image pair lists (Y, X).

Mask Image (Y). In order to generate the mask image, we must first define the meaning of this mask. As we expand before, the designer draws this mask to define *Game-Play Area*. In our case, we define *Game-Play Area* as an area containing space for placing enemies, items, events and so on. It does not mean a room, because after the designer draws a large game area, we want to make the neural network automatically divide the area into small rooms. In conclusion, the game-play area defines a high-intensity empty room space.

After we select the channel of empty space (e.g. the ground layer), we must do some image processing operations to reduce the wall between the rooms and decorative objects (e.g. rocks). In practice, we use one level close operation and three level open operation to reduce paths, small holes and decorative objects. Then we use a sigma $= 5$ Gaussian filtering to blur the results. For validation, we use a manual mask image to test the network during the training progress. Figure 6 shows an example.

Label Vector Image (X). Each label is treated as categorical vectors rather than an integer. Therefore, in each epoch, the selected one-channel image is converted to a categorical vector image. Theme rendering will uses the maximum value channel to be the category. That is to say, the network can even produce negative output vector, and the theme renderer still produces results.

Training Details. We trained each epoch on 10000 selected label image from the training database, and the batch size is 250. We trained 1000 epoch for results. The discriminator's learning rate is 0.0001 and the generator's learning rate is 0.0005.

5.3 Connection CNN Training

We use the example of connection CNN to explain the idea of training CNN. There can be different usage of CNN to post-process the output from conditional GAN. One of the important CNNs is *Connection CNN*, which is used to connecting areas as much as possible. In conditional GAN, we removed small paths and blur empty spaces for the mask, which make the mask lacks small details. GAN can generate different areas like mountains, lands, waters based on the mask but it is hard to generate paths and connections. So we used another CNN to generate them. Like we show in Fig. 7, it takes the output from GAN

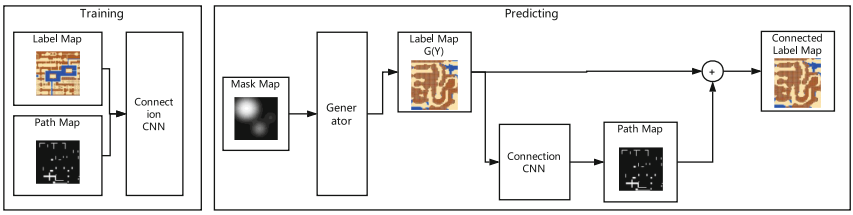

Fig. 7. *Connection CNN*, an example of post-processing CNN network. It takes a label map and a manually marked path map to train. And it takes conditional GAN's output as input to generate a path map. The path map is added to the generated result in order to connect areas.

and produces a one-channel image, which is called *Path Map*. These areas can be connected by adding a path map with manual control weights to the output ground channel.

We are not using the Pix2Pix framework since it is not a style transfer problem. The output of this CNN is not the same as the input. So we using an Encoder-Decoder U-Net CNN and use a binary cross-entropy as loss function to train this network directly. We manually marked the path in the game map to create training samples. Because the path map database contains smaller information and most area is black, we do not use image patches smaller than 64×64. We suggest limiting the value range of the path map to [0.01, 1.0], rather than [0.0, 1.0], which will provide more stable and faster training.

6 Authoring

We will talk about the authoring and theme rendering progress in this section. The authoring progress is shown in Fig. 1. This progress contains these steps:

- **Mask Capturing.** A web-based sketcher captures the designer's input and convert it into 0-1 mask image. It will be transferred to the back end to do the next steps.
- **Pre-Processing.** Pre-processing is applied to the mask contains blur and some other operation.
- **Network Processing.** Each network processes the previous output. The first network is always conditional GAN. And then it is post-processed by other networks like connection CNN.
- **Post Processing.** The results will be post-processed by traditional methods to avoid some strange results. Also, it combines outputs of all networks. This step will output a multi-channel label map.
- **Theme Rendering.** This process will select a theme renderer to generate images or 3D scenes based on the output label map.

So usually the designer's working progress is like this:

1. Draw a mask
2. Watch the theme renderer's result, which is sent to the website in 1–2 s.
3. Change some part of the mask.
4. Back to step 2.

6.1 Network Processing and Combine

After getting the mask image from designer, we convert that into floating-point format and apply a blur kernel on that. Then the blurred image is sent to conditional GAN to obtain an initial label map output, which will be sliced into different channels and used by other networks.

There are two different kinds of label maps, one is raw data from conditional GAN, whose value range much larger than $[0.0, 1.0]$, and the other is a clamped map. Which is these will be selected depends on the type of post-processing CNN. In our example, the connection CNN will use the range of values in $[0.0, 1.0]$. This value can be scaled the by divide the maximum number of all these values, or can be directly clamp in $[0.0, 1.0]$, which is our choice. However, the first method may show more information about the prediction info from conditional GAN because the ratio remains unchanged. Figure 8 compares the results before and after the connection CNN. The connection CNN take 'Ground' channel from output, and use that to generate a path map. We usually multiply the path map with a controlling weight and add it directly into 'Ground' output channel. The output from connection CNN sometimes contains a large number of writing areas, which is definitely not path. So we used an operation to remove this parts:

(Matrix B is 3×3 identity matrix.)

$$\begin{aligned} C &\leftarrow A \ominus B \\ A &\leftarrow Clamp(A - C, 0.0, 1.0) \\ A &\leftarrow Blur(A \oplus B) \end{aligned} \tag{3}$$

A is the output from path CNN, and B is the operation kernel.

The controlling weight is used to control the probability of paths. We used 100 and clamp them back to $[0.0, 1.0]$, which provided a very clear boundary for the path.

Fig. 8. This image shows one example of original output from conditional GAN and the connection CNN processed result. The first image is the original output. The second image is the path image produced from connection CNN, the third image is the connected areas. We can see the first area is not connected, and the final result is connected. The first and the final image is rendered by 2D height field theme renderer.

6.2 Theme Renderer

A theme renderer takes the final output label map as input, and generates an image or a 3D scene as output. At present, we have implemented a 2D theme renderer similar to Zelda style, a 2D height field theme renderer and a 3D terrain renderer. The 2D Zelda style theme renderer is used as a debugger to visualize each label. This theme renderer generates label map images. The 2D height field theme renderer uses a more detailed tile to draw and calculates the height of the mountain. Finally, we use World Machine to generate a complex terrain based on the height field to show it can be used to generate 3D game scenes.

Given a 2D height map from output without height information, we can calculate the height result by the Algorithm 1.

Algorithm 1. Calculate height result

set $H \leftarrow 0.0$
set $A \leftarrow Map_{mountain\ \ channel}$
while A has non-zero value **do**
  $H \leftarrow H + A$
  $A \leftarrow Binary\_Erosion(A)$
end while

A 3D theme renderer can take one or more channels to build a 3D scene. For example, it could take an output image from 2D height map renderer and build a 3D terrain by applying more operations like modify and erosion, as shown in Fig. 9.

7 Results and Discussion

The training database building was implemented in Python. The network has been implemented in Keras [4] based on TensorFlow backend. The training was performed on an Nvidia GeForce GTX 1080Ti with 11 Gb memory. We used a web-based designer front end based on HTML5 canvas and Ajax. However, since we can send JSON post from any application, so it can be replaced into others. The interactive designing system runs on a standard desktop computer with an Intel Core i5 CPU clocked at 3.00 GHz and with an Nvidia GeForce GTX 1050Ti. We use world machine 2 to build 3D terrain in order to show the 3D scene.

In this section, we will introduce the usage of this system, show the results of cGAN, WGAN, CNN, and the performance of our system. We will also talk about the limitation and the future works. Also we show the results in Figs. 13 and Fig. 14.

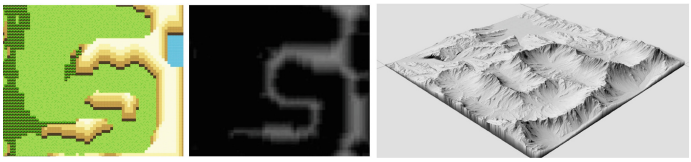

Fig. 9. A 3D scene theme renderer can use height map output from 2D height field renderer and then build a 3D scene. First image shows the 2D height field theme renderer's output, the middle image shows the height map dumped from the output. We build an algorithm in World Machine to produce the right image which shows a detailed terrain based on the height map. We do not use the path map to affect the height map in this case since we can consider the path map as a cave.

7.1 Usage

According to our goal, designers can use the system as a quick sketch game map designer. According to the masked image of the game area of the designer, the system can be used to generate complex game maps with multiple types of areas. After iteration, when the designer thinks it is good enough, the system can export different label channels to images and use them in other procedural generation systems to produce the final complex game scenarios. Because the steps from the label channel image to the final complex game scenario take time, our system allows designers to iterate quickly before making the final step.

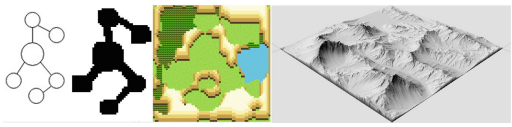

Fig. 10. Designers can use our system to fast iterate the map design. They think considering the map as a connected graph of the game-play areas, then use a sketch to draw this graph, and the network produces a 2D game map. If they decided, another procedural system produces a detailed 3D game scene, which may take more time. In this case, it took 1 min 8.8 s to generate.

We choose to use *Game-play Area* instead of terrain sketch like [13], because we think for game designer especially level designer, are more likely to view the entire game map structure as a connection map. They want to define game-play areas and connect them, and then they want to view game maps based on this structure. We show this progress in Fig. 10. However, for game artists like level artists, [13] is definitely a good solution to build complex terrain.

7.2 Design Pattern Learned by Network

We hope that our neural network can learn the design pattern from the given examples. In our case, maps of Legends of Zelda actually are rooms connected

with small path and holes. In each room, there are enemies, NPCs and items. This means the map is actually not a realistic style map from the real world but a designed map based on the game-play need. Usually, designers want to design maps which encourage players to explore, fight or rest. This requires that the map divided into individual rooms. Each room has its own functions. For example, a fighting area contains an enemy, or a safety room contains recovery items and saving points. Then the rooms are connected with other rooms by small holes or narrow paths. Enemies usually not pass these connections. It reduces the complexity of enemy AI and gives the player a chance to recover. The network has successfully learned this pattern and produce the same feature. We can see the rooms in the generated map like we show in Fig. 11. Sometimes, the network may even produce a "secret area" of a narrow door.

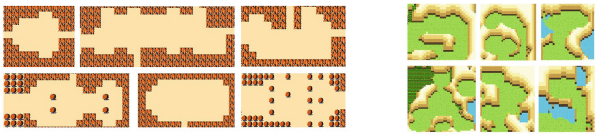

Fig. 11. Images in the first two raw are from maps of Legend of Zelda. It shows the room based design pattern. Rooms are individual areas connected by narrow paths. Images in the 3 and 4 row are the results of our network, which shows it successfully learned the pattern. We can recognize the rooms in these maps and they are connected by generated paths from conditional GAN and connection CNN.

7.3 Label Based Procedural Generation

We choose to generate label maps instead of results directly. Here are the reasons:

Training Stable. In our test, according to our label-based solution, it is really difficult to get good results for images generated directly from conditional GAN at 1024×1024 resolution. This is because the network needs to go deep enough to understand the meaning of each tile first, and then the relationship between the tiles. In fact, the design patterns which we want the network to learn is the relationship. Reducing the size and number of blocks will make the training remarkably stable and achieve better results.

Interact with Traditional Procedural Solution. There are many kinds of image-based procedural solution for generating game assets. By exporting specific kind of labels as an images, more complex scene can be generated. For example, we can use world machine 2 to generate terrain based on mountain label image. Or we can generate foliage based on tree label image. The label map is a guide for other procedural methods. This avoids networks to generate a huge map with complex details because it is tough.

Instead of generating the result label map in one network, we choose to use multi-networks. The label is a public data sharing by each network. Some networks can even produce additional labels. In our test, the more label kinds, the more unstable the network is, so it is better to reduce the label kinds and generate more labels step by step.

7.4 Comparisons

In this section, we will compare our methods with other methods in different ways, like network architecture and other traditional or neural network procedural solutions. We also test the differences caused by the number of label kinds.

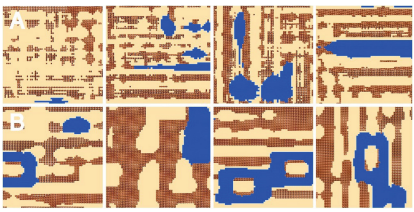

Fig. 12. Comparison between WGAN and CGAN. Upper raw contains results from WGAN output. Lower raw contains results from CGAN output.

Network Architecture. We tested the network with WGAN solution and conditional U-Net GAN solution. In fact, in the same number of label kinds, WGAN can also produce receptive results. We show the comparison in Fig. 12. The results look repeat without human control input. Also, we tested the encoder-decoder framework for DCGAN without U-Net. The control of input image looks little effect on the final results.

Other Traditional Methods. There are many other procedural ways to generate game maps or scenes. Methods without a neural network can be seen in [5]. Two most popular ways are as follows:

- **Grid Dungeon Generation** Generate a set of box rooms, and them move them until there is no collision. Then using paths to connect these rooms.
- **Cellular Automata** Using cellular automata to generate rooms with more randomize and complex dungeons.

Images for these methods can be seen on [2] and [3]. Our methods can generate rooms contains hard edges like the first method or rough edges like the second method. And almost both of these methods need manually define generation rules, but the difference lies in the type of rules. Directly controlling the generation results is not easy in these methods.

There are also industry software. One of them is Dungeon Architecture [1], which is a hybrid of grid dungeon generation and marker-based generation. It first generates dungeons based on the first method and then generates markers by human-designed rules. This software can produce a good result for dungeons. However, if build complex outside world, it needs a highly skilled designer to code for complex generation rules. This inspires the idea of label-based generation.

Neural Network-Based Methods. There are also many neural network-based methods. One of that is [13], which inspires us for using conditional GAN and U-Net framework. This method mostly focuses on generating terrain based on user's input image. This image defines the rough sketch of terrain. Then neural networks produce the large and detailed terrain shape. Our approach does not directly focus on terrain generation but can generate more than that. And also our input is not the rough sketch but the game-play area. We consider the usage is for game map designer to iterate the map fast, and other methods are for level artist to make terrain. And finally their output is a 3D terrain model but ours is an abstract label map for using in other processes.

7.5 Performance

Database and Training. We build the database convert from Legend of Zelda 1 on NES. The database contains 23040 64×64 samples, 3920 32×32 samples and 980 16×16 samples. All of these samples are scale into the same 64×64 size. The database is prepared in 30 min 20 s.

Our training label map is 64×64 resolution containing 5 channels. We use the integer format to store the label value and expand dynamically during each training epoch. The training is in 16 h with 1000 epoch in GTX 1080 Ti.

Run-Time Performance. In each update request, our system's performances are as follows:

- **Request Prepare Time: 0.004 s.** Contains the time getting from the designer front end and translate into the input image with blur filter.
- **conditional GAN Time: 1.468 s.** From conditional GAN which takes input and produces an output label map in 64×64 contains 5 channels.
- **connection CNN Time: 0.144 s.** From taking the output from conditional GAN and produce a path map in 64×64 contains one channel.

7.6 Limitations and Future Work

Our system still has some limitations, like the follows:

1. Areas inside the small rooms does not contain many details. It lacks elements. We think this can be fixed by another decorator CNN to produce decorators like rocks and trees inside the empty rooms.

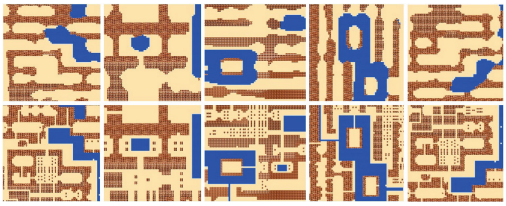

Fig. 13. This image shows the conditional GAN output and the ground truth. For each column, the upper one takes the lower one's game-play area, and generate an output, the lower one is original map clips from train database.

Fig. 14. This is more generated results based on the randomly chosen input. It shows even regular input can generate complex results with different area types. We can also recognize a room based map design pattern in these results.

2. 3D theme renderer cannot be produced in real time. The most time cost thing is the traditional erosion operation to produce complex details for terrain map. This can be fixed by integrating a system like [13].
3. Side running games like Super Mario do not fit by our cGAN system. It cannot be described easily with game-play areas. However, our WGAN system can still support this type. How to integrate human control into this type of game is still a question in our system.
4. By using some real-time methods like [11,17,18,20,21], the system's performance can be improved. Also, many new network architectures can be used to in this system like [14,15,29].

7.7 Conclusion

This paper introduces a framework for procedurally generating game maps based on human control and given example samples. The framework can learn map design pattern from given examples, take human input control into account, and generate an abstract label map with the same pattern type. This process can be done in real time.

Designers treat the game map design as draw game-play areas and connect them, and then will get images contains ground, mountains, trees and water areas. Finally, we introduce how to use Theme Renderer to convert the label map into 2D maps and 3D scenes, which shows how this system interacts with transitional procedural generation systems to produce high detailed results. We have explained the details about the neural network used in this system. And

we hope to show the potential of using Deep Learning methods in procedural content generation.

References

1. Content detail
2. Create a procedurally generated dungeon cave system
3. Generate random cave levels using cellular automata
4. Keras: The python deep learning library
5. Procedural content generation wiki
6. Procedural generation, December 2017
7. Arjovsky, M., Chintala, S., Bottou, L.: Wasserstein gan. arXiv preprint arXiv:1701.07875 (2017)
8. Ashlock, D., Mcguinness, C.: Automatic generation of fantasy role-playing modules. In: 2014 IEEE Conference on Computational Intelligence and Games (2014)
9. Cordonnier, G., Galin, E., Gain, J., Benes, B., Guerin, E., Peytavie, A., Canie, M.-P.: Authoring landscapes by combining ecosystem and terrain erosion simulation. ACM Trans. Graph. **36**(4), 134 (2017)
10. Gain, J., Marais, P., Straßer, W.: Terrain sketching. In: Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games, pp. 31–38. ACM (2009)
11. Gentile, C., Li, S., Kar, P., Karatzoglou, A., Zappella, G., Etrue, E.: On context-dependent clustering of bandits. In: Proceedings of the 34th International Conference on Machine Learning, vol. 70, pp. 1253–1262. JMLR. org (2017)
12. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y.: Generative adversarial nets. In: Advances in Neural Information Processing Systems, pp. 2672–2680 (2014)
13. Guérin, É.: Interactive example-based terrain authoring with conditional generative adversarial networks. ACM Trans. Graph. (TOG) **36**(6), 228 (2017)
14. Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., Meger, D.: Deep reinforcement learning that matters. In: Thirty-Second AAAI Conference on Artificial Intelligence (2018)
15. Imani, M., Ghoreishi, S.F., Allaire, D., Braga-Neto, U.M.: MFBO-SSM: multi-fidelity Bayesian optimization for fast inference in state-space models. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, pp. 7858–7865 (2019)
16. Isola, P., Zhu, J.-Y., Zhou, T., Efros, A.A.: Image-to-image translation with conditional adversarial networks. arXiv preprint arXiv:1611.07004 (2016)
17. Kar, P., Li, S., Narasimhan, H., Chawla, S., Sebastiani, F.: Online optimization methods for the quantification problem. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1625–1634. ACM (2016)
18. Korda, N., Szörényi, B., Shuai, L.: Distributed clustering of linear bandits in peer to peer networks. In: Journal of Machine Learning Research Workshop and Conference Proceedings, vol. 48, pp. 1301–1309. International Machine Learning Society (2016)
19. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)

20. Li, S.: The art of clustering bandits. Ph.D. thesis, Università degli Studi dell' Insubria (2016)
21. Li, S., Karatzoglou, A., Gentile, C.: Collaborative filtering bandits. In: Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 539–548. ACM (2016)
22. Mirza, M., Osindero, S.: Conditional generative adversarial nets. arXiv preprint arXiv:1411.1784 (2014)
23. Olsen, J.: Realtime procedural terrain generation (2004)
24. On, C.K., Foong, N.W., Teo, J., Ibrahim, A.A.A., Guan, T.T.: Rule-based procedural generation of item in role-playing game. Int. J. Adv. Sci. Eng. Inf. Technol. **7**(5), 1735 (2017)
25. Qi, Y., Shen, X.-K., Duan, M.-Y., Cheng, H.-L.: A method of rendering clouds with perlin noise [j]. Acta Simulata Systematica Sinica **9**, 023 (2002)
26. Radford, A., Metz, L., Chintala, S.: Unsupervised representation learning with deep convolutional generative adversarial networks. arXiv preprint arXiv:1511.06434 (2015)
27. Ronneberger, O., Fischer, P., Brox, T.: U-Net: convolutional networks for biomedical image segmentation. In: International Conference on Medical Image Computing and Computer-Assisted Intervention, pp. 234–241. Springer (2015)
28. Schaal, J.: Procedural terrain generation. A case study from the game industry. In: Game Dynamics, pp. 133–150 (2017)
29. Wang, C., Bernstein, A., Le Boudec, J.-Y., Paolone, M.: Explicit conditions on existence and uniqueness of load-flow solutions in distribution networks. IEEE Trans. Smart Grid **9**(2), 953–962 (2016)
30. Zhou, H., Sun, J., Turk, G., Rehg, J.M.: Terrain synthesis from digital elevation models. IEEE Trans. Vis. Comput. Graph. **13**(4), 834–848 (2007)