Online Level Generation in Super Mario Bros via Learning Constructive Primitives

DOI:
10.1109/CIG.2016.7860397

Document Version
Accepted author manuscript

Online Level Generation in Super Mario Bros via Learning Constructive Primitives

Peizhi Shi and Ke Chen
School of Computer Science
University of Manchester
{shipa, chen}@cs.manchester.ac.uk

***Abstract*—In procedural content generation (PCG), how to assure the quality of procedural games and how to provide effective control for designers are two major challenges. To tackle these issues, this paper exploits the synergy between rule-based and learning-based methods to produce quality yet controllable game segments in *Super Mario Bros* (SMB), hereinafter named *constructive primitives* (CPs). Easy-to-design rules are employed for removal of apparently unappealing game segments, and subsequent data-driven quality evaluation function is learned based on designer's annotations to deal with more complicated quality issues. The learned CPs provide not only quality game segments but also an effective control manner at a local level for designers. As a result, a complete quality game level can be generated online by integrating relevant constructive primitives via controllable parameters. Extensive simulation results demonstrate that the proposed approach efficiently generates controllable yet quality game levels in terms of different quality measures.**

I. Introduction

Procedural content generation (PCG) is of great interest to game design and development as it generates game content automatically. In PCG research, a number of methods have been proposed for game content generation in different game genres [1]. *Super Mario Bros* (SMB), a classic 2D platform game, has become a popular test bed for PCG-related research [2], [3]. In this platform game, a player runs from the left side of the screen to the right side, fights enemies, and rescues *Princess Peach*. SMB has a number of different game elements (e.g., enemies, tubes, and cannons) that can be used in PCG. In recent years, several SMB level generators have been developed for *level generation track of the Mario AI Championship* [4] as well as presented in publications [5]–[11].

Although wide varieties of PCG techniques have been proposed, there are still some open challenges faced by SMB level generation as well as generic PCG techniques. One challenge in PCG is how to assure the quality of procedural content. Procedural levels sometimes contain unplayable structures [1], aesthetically unappealing items [8], unexplainable difficulty spikes [12] and unreachable resources [13], which could result in negative gameplay experience. Another challenge is how to effectively control the geometrical features (e.g., coordinate of each enemy and tube) and some properties of procedural levels (e.g., linearity [14] and density [6]). In general, a game designer has to encode the desired properties in handcrafted rules (e.g., theory-driven evaluation functions) to control the procedural levels [12], that could slow down the level generation [13].

This paper presents an alternative approach to address the aforementioned issues. Motivated by the learning-based PCG (LBPCG) framework [16] and other existing works, we explore the content space in SMB from a different perspective by taking short game segments into account. To address the quality assurance issue, we exploit the synergy between rule-based and learning-based methods. Easy-to-design rules are employed for removal of apparently unappealing game segments, and then a data-driven evaluation function is constructed based on designer's annotations about the quality of game segments. Once the data-driven evaluation function is constructed, we use it along with the aforementioned rules to produce high quality game segments, hereinafter named *constructive primitives* (CPs). Those CPs not only provide quality game segments but also enable to control the geometry and the level properties effectively. As a result, a complete quality game level can be generated online by integrating relevant CPs together via controllable parameters. Experimental results demonstrate that our data-driven evaluation function could implicitly encode multiple quality-related criteria, and improve the quality of procedural level with respect to different quality measures. Results also show that our generator is capable of generating procedural levels of desired properties online.

The main contributions of the paper are summarized as follows: a) a novel approach to producing quality yet controllable game segments or CPs in SMB; b) a controllable online level generator based on CPs; and c) a thorough evaluation of our proposed approach.

II. Learning Constructive Primitives

In this section, we first describe the motivation underlying our approach and then present our approach to producing constructive primitives (CPs) in SMB.

A. Motivation

By a close look at existing SMB level generators, we observe that the content space on all the complete procedural levels is huge. As there are an enormous variety of combinations among game elements and structures at procedural levels, an approach working on such content space inevitably faces a greater challenge in managing quality assurance and generation efficiency in PCG. Nevertheless, a complete procedural level in SMB can be decomposed into a number of segments as evident in [7]–[9]. Partitioning levels into fixed-size game pieces permits us to decompose the level design problem [15]. As a result, all the possible segments form a new content space of lower complexity. We believe that it is less difficult to understand the properties and quality of short game segments and hence the use of those segments as building blocks would facilitate tackling aforementioned non-trivial issues in SMB.

For quality assurance, there are generally two methodologies in developing such a mechanism in PCG [1], [16]: deductive vs. inductive. To adopt the deductive methodology, game developers have to understand the content space fully and know how to formulate/encode their knowledge into rules explicitly. In the presence of a huge content space, however, it would be extremely difficult to understand the entire content space. Thus, less accurate (even conflicted) rules might be used in PCG, which could generate low quality games. Nevertheless, we observe that some rules are easy to design/identify while a complete set of rules for evaluating the content quality are hard to handcraft. For example, overlapped tubes in SMB is unacceptable and can be easily detected with a simple rule. On the other hand, a learning-based PCG (LBPCG) framework [16] was recently proposed where an inductive methodology, i.e. learning from data, was advocated for quality assurance. As game content is observable but less explainable, it is easier for game developers to make a judgement on quality for a specific game by applying their knowledge implicitly than to encode their knowledge into rules or constraints [8]. Thus, the LBPCG suggests that a quality evaluation function should be learned from data annotated by game developers. Hence, a hybrid approach to quality assurance would allow us to exploit the synergy between rule-based and learning-based methods.

For controllability, game developers usually encode desired properties (e.g., linearity, leniency) into theory-driven evaluation functions (e.g., [5], [7]). Then, game level of desired properties can be generated via generate-and-test method, which is not efficient. We observe that it could provide more efficient and effective control for designers if they directly select game from desired region of content space instead of using theory-driven evaluation functions to explore the content space.

With the motivation described above, we propose a hybrid approach to producing CPs, quality yet controllable game segments, in SMB. Fig. 1 illustrates the main steps of our approach. First of all, game developers choose a region of interest from the entire content space via controllable parameters. Then game segments in the region of interest are evaluated by a set of easy-to-design handcrafted conflict resolution rules and the subsequent data-driven quality evaluation function that deals with more complicated quality issues. Survivals of game segments become CPs.

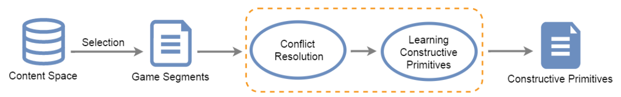

Fig. 1. The constructive primitive (CP) generation process for SMB.

B. Content Space

We observe that it is sufficient to cover rich yet diverse types of levels by using a game segment of 20 in length and 15 in height. Some typical game segment instances are illustrated in Fig. 2.

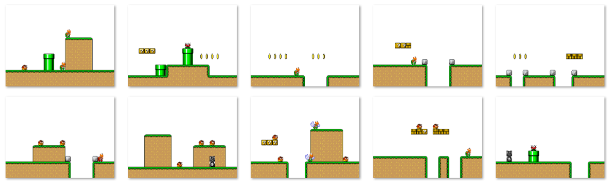

Fig. 2. Game segment instances.

The SMB content is naturally specified by a 2D grid similar to an image. However, this leads to a 300-dimensional content space in our case where there are a lot of redundancies, e.g., the uniform background. In this paper, we employ a list of design elements as our content space representation where a design element refers to an atomic unit used in a procedural level generation, e.g., enemy, boxes, coins, cannon, and gap. By using this representation, we can not only specify the content space concisely but also gain the direct controllability on low-level content features, e.g., coordinates of enemies and coins. As listed in Table I, 85 controllable features are employed in our representation. Such representation is similar to the previous work [5], [6]. In this representation, x, y, $width$, $height$, and $type$ refer to the x coordinate, y coordinate, width, height, and type of each design element, while w_{before} and w_{after} refer to width of the platform before and after each tube/cannon. Among these features, $types$ of gap, tube, boxes and coins are nominal features, while the rests are ordinal features. In our content space, the design elements in each type are sorted in decreasing order along x dimension.

TABLE I. CONTENT FEATURES

ID	Description
1	$height$ of initial platform
2	number of gaps
3 - 11	x, $width$ and $type$ of the 1st - 3rd gap
12	number of hills
13 - 18	x, $width$ and $height$ of the 1st and 2nd hill
19	number of cannons
20 - 34	x, y, $height$, w_{before} and w_{after} of the 1st - 3rd cannon
35	number of tubes
36 - 53	x, y, $height$, w_{before}, w_{after} and $type$ of the 1st - 3rd tube
54	number of boxes
55 - 62	x, y, $width$ and $type$ of the 1st and 2nd boxes
63	number of enemies
64 - 78	x, y and $type$ of the 1st - 5th enemy
79	number of coins
80 - 85	x, y and $width$ of the 1st and 2nd coins

While design element parameters in Table I have a wide range that specifies a huge content space, we confine our concerned content space to a non-trivial region of the content space by setting the maximum number of gaps, hills, tubes, cannons, boxes, coins, and enemies appeared in a game segment are 3, 2, 3, 3, 2, 2, and 5 respectively. These design decisions are made based on our game design knowledge. Consequently, there are roughly 9.72×10^{37} game segments in our content space. This content space should be sufficient for generating content with a variety of geometrical features, level structures and difficulties required by SMB.

The content space defined in Table I is an explicit controllable space, which means that game designers can effectively control the properties of game segments by specifying the desired region of content space. For instance, a pure linear segment can be generated if designers set the number of hills,

y coordinate of tubes and cannons as zero, and set the rest of controllable features on random; a mountainous segment can be generated by setting the number of hills as 1 or 2, and set the rest of controllable features on random.

C. Conflict Resolution

In our content space, there are quite a number of game segments that contain conflicting design elements. For instance, "…*Tube(5,0,2,0,0,normal)*…*Cannon(5,0,4,0,0)*…" represents a game segment of at least one tube and one cannon. The x, y, $height$, w_{before}, w_{after}, and $type$ of this tube are 5, 0, 2, 0, 0, and $normal$ respectively, while the x, y, $height$, w_{before}, and w_{after} of this cannon are 5, 0, 4, 0, and 0 respectively. We can see from above description that their x coordinates are same. Thus, the cannon and tube are overlapped together and this conflicting situation makes the segment aesthetically unappealing.

To address this issue, we adapt a class of rules presented in [5], [6] for our requirement. Whenever two design elements in a game segment are overlapped together, this game segment is discarded during CP generation. In our approach, gap, enemy, tube, cannon, boxes, and coins are not allowed to be overlapped with each other, while hills of different heights can be overlapped together. In addition, enemy/tube/cannon can be overlapped with hills.

D. Learning Constructive Primitives

After filtering out those obviously unappealing game segments, the tailored content space still contains a lot of low quality segments, e.g., segments of unplayable or aesthetically unappealing structures, segments of unreachable or unbalanced resources, and segments that lack a sense of progression. Inspired by the LBPCG work [16], we would learn a quality evaluation function from annotated game segments to remove unplayable/unacceptable segments. To carry out this idea, a binary classifier is trained where its input is the 85D feature vector of a game segment and its output is a binary label that predicts the quality of a game segment. Binary classifier rather than multi-class classifier is chosen since it is easier for annotator to make binary decision about segment quality and avoid using knowledge explicitly. Game segments labeled as positive are CPs and would be used for online level generation described in Sect. III.

To establish a data-driven evaluation function, training examples are required but have to be provided from game developers. As the tailored content space is still huge, it is infeasible to annotate all possible games in this content space. To keep the content space manageable, a proper sampling can be applied to achieve a much smaller data set of the same properties as the content space. Motivated by the success in the LBPCG work [16], we conduct clustering analysis on the data set and further employ active learning based on the clustering results to create a data-driven quality evaluation function (or classifier). Clustering algorithm is used since it can get a better estimation of data distribution, while active learning is chosen to minimize a game developer's efforts in data annotation. In summary, this CP learning process is depicted in Fig. 3.

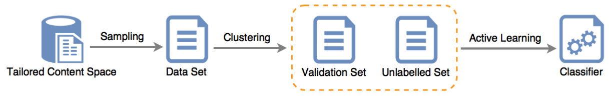

Fig. 3. The constructive primitive learning process.

1) Sampling: For sampling, we apply the simple random sampling (SRS) with replacement [17] to the tailored content space for a manageable data set. In contrast to other sampling techniques, SRS is an unbiased sampling technique which can ensure that each game segment in this content space has the equal probability of selection. In addition, it can handle unknown data distribution without any prior knowledge since we do not know the distribution of content space. As a result, we randomly set all the controllable features in the tailored content space to form a data set.

In sampling, an important question is how large a sample should be. Large sample size can guarantee that sample is representative enough, but can also result in increased human work. The size of our data set is determined via the sample size determination (SSD) algorithm suggested in [17] since this algorithm is designed for simple random sampling. According to SSD algorithm, necessary sample size can be calculated as follow:

$$n \approx \frac{z^2\delta^2}{d^2} = \frac{1.96^2 \times 49.4490}{0.10^2} = 18996.32 \approx 19000 \qquad (1)$$

where z is the z-score 1.96 for 95% confidence interval, and d is 0.10 which refers to a small allowable difference. These values are suggested in typical SSD algorithm. δ^2 is 49.4490 which refers to maximum variance of content space among different features. The value of δ^2 is determined using an *engineering guesstimate* [18]. With the theoretical justification, the SSD can decide the size of a sampled data set without loss of non-trivial information. By applying the SSD to our tailored content space, it is suggested that a data set of 19,000 games should be sufficient.

2) Clustering: We apply the CURE algorithm [19] on the sampled segment set for clustering analysis since this hierarchical clustering algorithm can deal with data set with unknown data distribution and discover the clusters of different sizes. We can achieve good clustering results without any prior knowledge of data distribution within our data set [20]. There are four parameters in CURE algorithm: the number of clusters, sampling rate, shrink factor, and the number of representative points. By using the dendrogram tree achieved, the number of clusters is automatically decided based on the longest k-cluster lifetime [21]. The rest of parameters are set to defaults suggested in [19]; i.e., 2.5% for sampling rate, 0.5 for shrink factor and 10 representative points, respectively. Due to the existence of two different feature types, i.e. nominal and ordinal, we employ the mixed-variable distance metric [20] in the CURE. After clustering, we found 106 clusters from this sampled data set. Game segments within same cluster tend to have similar structures (e.g., same number of design elements), while segments in different clusters may look different. The clustering results would be used to facilitate active learning.

3) Active Learning: For binary classification, there are two error types: *false negative* (type-I error) where a high quality segment is misclassified as low quality and *false positive* (type-II error) where a low quality segment is misclassified as high quality. Obviously, a type-II error could result in a catastrophic effect while a type-I error simply shrinks the content space slightly. As a result, we formulate our classification as a cost-sensitive learning problem where the type-II error incurs a higher cost. By looking into several state-of-the-art classification techniques, we found that the weighted random forests (WRFs) [22], a cost-sensitive oblique random forests [23] classifier, fully meet our requirements for active learning. Random forests are an ensemble learning method for classification and regression by training different decision trees based on different subsets of the data. Such classifier can handle data set with different feature types, and allows us to know which feature is more important during training. WRFs is a cost-sensitive version of random forests in which the class weights are taken into consideration. Such algorithm is easy to incorporate into typical random forests, and allows us to control the weights of two types of errors easily. In our work, the parameters of WRFs [22] are set via validation as follows: 2:1, 50, 5, 10, and 9 for the cost ratio, the number of trees, the number of combined features, the number of feature groups selected at each node, and depth of trees, respectively.

After clustering, a small number of segments are selected from each cluster to form a validation set in order to evaluate the generalization performance of a classifier during active learning. The number of segments selected from each cluster is proportional to the cluster size. Totally, there are 800 segments in the validation set. One of author annotates each game segment in the validation set by visual inspection. In general, it takes us less than five seconds to annotate a game segment.

During active learning, we randomly choose 100 segments and annotate them via visual inspection to train the initial WRFs. In each iteration, we find 100 segments of the highest uncertainty scores, defined by $s_i = 1 - P(\hat{y}|x_i)$ where $\hat{y}$ is the predicted label of segment x_i, and $P(\hat{y}|x_i)$ is the probability of this prediction, and annotated them to be examples for re-training WRFs. The active learning stops when the accuracy of WRFs on the validation set no longer increases. Although there are other active learning techniques, our active learning algorithm based on uncertainty sampling is efficient to handle a data set of of 19,000 points. Such algorithm is summarized in Algorithm 1.

Once the evaluation function is learned, we use it along with the rules described in Sect. II.C to produce CPs in a generate-and-test way, and a combination of proper CPs via controllable parameters leads to an online procedural level generator as described in next section.

III. Online Level Generation

As described in Sect. II, CPs provide quality building blocks and hence lumping them together can easily lead to a procedural level of aesthetically appealing content with a path between entrance and exit. In SMB, there are a variety of procedural levels that can be categorized based on a number of properties, e.g., density [6], leniency [14], and linearity [14]. As our CPs are represented by design elements, we can generate a procedural level of pre-setting property via the corresponding controllable level generation parameters.

Algorithm 1 Active Constructive Primitive Learning

```
Input: Sampled data set $U$ and clustering results on $U$.
Output: WRFs binary classifier.
Initialization: Based on the clustering analysis results,
create a validation set $V$ of 800 examples.
Active Learning:
Annotate 100 segments randomly selected from $U$ via visual
inspection to form a training set $L$. Train WRFs on $L$ to
obtain an initial binary classifier.
repeat
  for all $x_i \in U$ do
    Label $x_i$ with the current WRFs.
    Calculate the uncertainty score $s_i$ of $x_i$.
  end for
  Annotate 100 segments of the highest uncertainty score
  in $U$ to form a new training set $L$.
  Re-train the WRFs with the examples in $L$.
until The overall accuracy on $V$ does not increase.
return Classifier WRFs.
```

Motivated by the previous works [6], [14], we employ three controllable level generation parameters, i.e., density, leniency, and linearity, to generate a variety of levels online. The density controls the complexity of geometrical structures, e.g., a high density leads to many overlapping hills. The leniency decides the level difficulty in gameplay; intuitively, a high leniency results in an easy-to-play level. The linearity is yet another parameter that ensures there is a linear structure in a generated level; a large value leads to a level of highly linear structures. Each level generation parameter is set to $\{1, 2, 3\}$, and carried out by setting the proper values to relevant content features in CPs as defined in Table II. It is worth stating that linearity may conflict with density. Hence, we stipulate that the density and linearity parameters cannot be used together.

TABLE II. Parameters used in game generation

Parameter	Value	Description
Leniency	1	number of enemies ≥ 2; number of gaps ≥ 1
	2	number of gaps ≤ 2; $width$ of gaps < 3; $1 \leq$ number of enemies ≤ 3; enemies without wings
	3	number of enemies ≤ 1; number of gaps ≤ 1; number of cannons $= 0$; $width$ of gaps < 3; no turtle enemy
Linearity	1	$0 \leq$ number of hills ≤ 2; $height$ of the first platform $= 2$
	2	y coordinates of tubes and cannons $= 0$; number of hills ≤ 1; number of hills ≤ 1; $height$ of the first two platforms $= 2$
	3	y coordinates of tubes and cannons $= 0$; number of hills $= 0$; number of hills $= 0$; $height$ of the first three platforms $= 2$
Density	1	$0 \leq$ number of hills ≤ 1
	2	$0 \leq$ number of hills ≤ 2
	3	number of hills ≥ 1; number of gaps ≥ 1

To generate a complete level, we first specify the desired values to controllable parameters that fix the values of relevant content features and set other irrelevant content features in game segments randomly. Thus, game segments of desired properties are generated, and evaluated by rule-based and data-driven quality evaluation functions. Survivals of game segments become CPs, and an iterative process is undertaken by merging the CPs of the specified properties together until reaching a pre-specified length.

As depicted in Fig. 4, our CP-based online generation algorithm first uses a generate-and-test method to produce CPs for quality assurance, and a complete procedural level is then constructively generated by sequentially lumping CPs of specified properties together via setting controllable parameters at a local level.

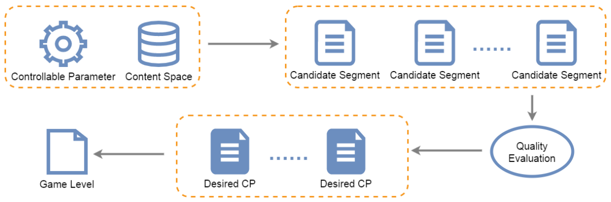

Fig. 4. Online procedural level generation.

IV. Experimental Results

In this section we report results in the CPs learning and level generation to study the implications and benefits of our data-driven evaluation function, and examine whether our generator is capable of generating controllable yet quality game level efficiently in terms of different quality measures. The game engine adopted in our experiments is a modified version of the open-source *Infinite Mario Bros* used in the *Mario AI Championship* [4], [24]. Our level generator that yields results reported in this section are publicly available on our project website[1].

A. Results on Constructive Primitives Learning

Based on the learning algorithm described in Sect. II, Fig. 5 illustrates the evolutionary performance of our active learning on the validation set, including types-I and -II error rates as well as their average, the *half total error rate* (HTER). From Fig. 5, it is observed that the active learning converges after 1100 data points.

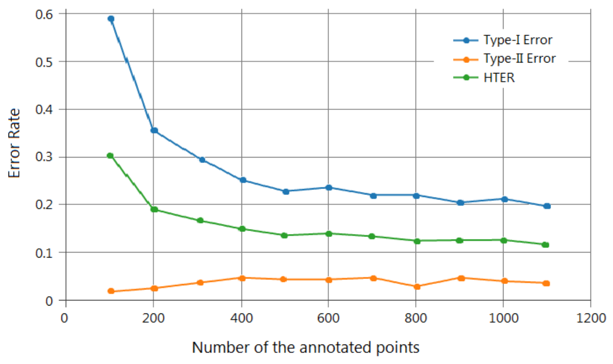

Fig. 5. Performance evolution on the validation set during active learning.

While the final HTER is around 11.67%, the corresponding type-I error rate is around 19.66%. It is evident from Fig. 5 that our cost-sensitive classifier performs well in minimizing the type-II error; the type-II error rate is approximately 3.69%. Among those segments that yield type-II error, about 0.74% segments contain unreachable resources and the rest segments consist of unexplainable difficulty spikes, imbalanced resources, and aesthetically unappealing structures. While such segments may result in negative gameplay experience, fortunately, none of unplayable segments in the validation set was misclassified.

Some correctly classified and misclassified instances are shown in Fig. 6. Segment instances in Fig. 6 (A) and (B) are positive examples, while instances in Fig. 6 (C) and (D) are negative examples. The label of each instance is determined according to our labeling criteria. For instance, the 1st example in Fig. 6 (C) consists of unplayable structure which does not allow a player to pass through. The 4th example in Fig. 6 (C) contains unreachable boxes. The 1st, 2nd and 3rd examples in Fig. 6 (D) consist of unbalanced resources and difficulty spikes. All the game elements (e.g., boxes, enemy, cannon, gap) are centralized in the middle of game segments, which will dramatically increase the difficulty in the middle of game segments. The 2nd example in Fig. 6 (C) contains simple structures which lack a sense of progression. The rest segments in Fig. 6 (C) and (D) consist of aesthetically unappealing combination since enemies, gap, boxes, tubes and cannons are arbitrarily lumped together without any meaning. In contrast, segments in Fig. 6 (A) and (B) contain playable structures, reachable resources and meaningful combination which capture an area of challenge and convey a sense of progression. Among all these instances, game segments in Fig. 6 (B) and (D) will yield type-I and -II errors respectively.

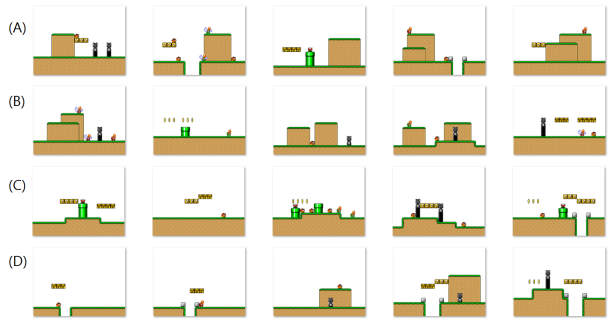

Fig. 6. Segment instances. (A) Correctly classified positive examples. (B) Segments leading to type-I error. (C) Correctly classified negative examples. (D) Segments leading to type-II error.

By analysing clustering analysis results, we found that segments located in same region of content space tend to be classified as same category. For instance, the 5th example in Fig. 6 (B) and rest examples in its cluster are classified as negative since most instances in this cluster are annotated as negative examples. During training, WRFs finds a single split that optimizes the information gain to classify these instances. Thus, some of them are misclassified. The 2nd example in Fig. 6 (B) is also misclassified due to same reason. In addition, segments of complicated data distribution are likely to be misclassified. For instance, the 5th example in Fig. 6 (C), 4th and 5th examples in Fig. 6 (D) share similarities since they both have boxes above the rock gap, but these instances are located in different regions of content space due to their representations. During training, WRFs has to find three different splits instead of one split to classify these segments. Thus, some of them will be misclassified since these splits cannot optimize information gain. The data distribution of this type of segments is complicated since these data points look similar from perspective of annotator, but spread through

[1]http://staff.cs.manchester.ac.uk/šhipa/mario.html

whole content space due to their representations. This is the main source of this type of errors. The 1st, 3rd, and 4th examples in Fig. 6 (B) are also misclassified due to this reason. Moreover, some "rare segments" are likely to be misclassified. The proportion of some types of game segments (see the 1st, 2nd and 3rd examples in Fig. 6 (D)) in whole content space is small. On one hand, those segments are located in several quite small regions of the content space, and unlikely to be selected in both training and validation set. On the other hand, WRFs uses greedy strategy to find an optimal split according to the split criteria. These "rare instances" maybe overlooked and then misclassified since these splits cannot optimize the information gain.

Experimental results demonstrate that our data-driven evaluation function can effectively eliminate low quality segments in terms of our labeling criteria. This argument is supported by the low type-II error achieved from active learning. In addition, this function can implicitly encode multiple quality-related criteria, which could avoid the design and use of multiple evaluation functions.

B. Quality Analysis

This section further examines the implications and benefits of our data-driven evaluation, and compares our approach to other generators in terms of quality. Unlike [8] that uses human subjects to evaluate their generator, we use different quality measures to compare our system with others since these metrics are capable of evaluating large numbers of levels objectively.

In this experiment, we use strategic resource control (f_s), area control (f_a), area control balance (b_a), and exploration (E) defined in [25] as our metrics for evaluation since they are domain-independent metrics which are verified by game developers. Among these metrics, f_s measures how close treasures (e.g., coins, boxes) are to enemies; f_a measures the placement of design elements (e.g., enemies, tube flowers, cannons) away from each other; b_a measures how balanced distribution of design elements; E evaluates the exploration efforts to reach the exit starting from entrance. Mathematical definitions of these metrics are defined in [25]. Apart from above measures, we also employ Peter Lawford's A* agent [24] in small state to evaluate the playability (P) of game levels since this agent can survive from most playable game levels no matter how difficult they are.

As we aim to examine the benefits of our data-driven evaluation function, we use our generator to generate two sets of levels. Each level in the first set is composed of randomly generated constructive primitives, while each level in the second set is composed of game segments which survive from conflict resolution rules but are rejected by data-driven evaluation function. For a thorough evaluation, we compare our approach with a number of SMB level generators, including Notch [3], Grammar Evolution (GE) [6], and generators developed for *level generation track of the Mario AI Championship* [4]. Each of level generators generates 100 procedural levels of 200 in width and 15 in height for evaluation in terms of aforementioned metrics. The level generation parameters used in ours are set randomly and others use their default settings. The E scores are normalized to the range of [0,1].

Table III illustrates the mean and the standard deviation of quality scores achieved from different level generators. We now use one-tailed Wilcoxon rank sum test to make a pairwise comparison between our approach (levels in set 1) and others with respect to different quality metrics. From Table III, it is evident that both ours (levels in set 1), GE generator, and Mawhorter's generator receive high f_s scores, which implies that treasures (e.g., coins, boxes) in these levels are close to enemies acting as their "guardians" [25]. In contrast, design elements (e.g., enemies, boxes, coins) in procedural levels of set 2 are arbitrarily lumped together without any reason, which leads to relatively lower f_s score. Regarding area control (f_a) and area control balance (b_a), our generator tends to generate levels with significantly higher f_a score than others and moderate b_a score, which implies that design elements (e.g., enemies, tube flowers, cannons) in our procedural levels are placed far away from each other, and have balanced distribution. For exploration (E), Mawhorter's generator [11] tends to generate levels with complicated structures and multiple paths, which leads to highest E score. In contrast, ours generate levels of low E score in comparison to others due to the initial content space defined in our approach. We found that our density parameter could increase E score since it controls the complexity of geometrical structures. Such result is not presented here since this experiment only examines the benefits of data-driven evaluation function instead of controllable parameters. For playability (P), both ours, Notch generator and Takahashi's generator achieve a high percentage of playable levels. The playability results of these three approaches are significantly better than the rests. Please notice that P score value is dependent on A* agent's gameplay performance and sampled levels. Procedural levels that A* agent cannot pass through are not necessarily unplayable.

TABLE III. AVERAGE QUALITY SCORE.

Generator	f_s	f_a	b_a	E	P
Ours (set 1)	0.34(0.11)	0.46(0.08)	0.48(0.07)	0.20(0.07)	1.00(0.00)
Ours (set 2)	0.29(0.10)	0.36(0.09)	0.45(0.07)	0.18(0.06)	0.92(0.27)
Notch	0.27(0.26)	0.24(0.19)	0.42(0.24)	0.12(0.05)	1.00(0.00)
GE	0.34(0.19)	0.41(0.12)	0.42(0.09)	0.17(0.05)	0.96(0.20)
Weber	0.29(0.05)	0.40(0.07)	0.43(0.06)	0.33(0.14)	0.28(0.45)
Shimizu	0.24(0.05)	0.29(0.04)	0.50(0.06)	0.32(0.07)	0.93(0.26)
Mawhorter	0.32(0.06)	0.42(0.07)	0.43(0.07)	0.44(0.19)	0.81(0.39)
Takahashi	0.23(0.09)	0.19(0.10)	0.54(0.15)	0.18(0.05)	1.00(0.00)
Baumgarten	0.18(0.05)	0.20(0.05)	0.53(0.06)	0.31(0.11)	0.86(0.35)

Extensive simulation results show that our data-driven evaluation could improve the quality of procedural games with respect to different quality measures, and our generator is capable of generating game levels with high f_s, f_a, and P scores.

C. Controllability

In this section, we evaluate the controllability of our level generator. In general, controllability can be reflected in the expressive ranges of procedural levels generated with different level generation parameter settings [7]. Expressive range refers to the range and variation of procedural levels according to an evaluation metric [14]. This paper uses *linearity* [14], *density* [6], and *leniency* [14] as our metrics since they can reveal global properties of game levels generated by a level generator. For linearity, we use the method suggested in [13] to find a line that fits the profile of a procedural level, and the coefficient of determination r^2 is used to estimate the degree of linearity. For density, we count the number of all possible standing positions in a game level [6]. For leniency, we assign a value to each type of game elements as same as used in [6], [7] (i.e., enemy: -1.0, gap, cannon or flower tube: -0.5, and powerup: +1.0). The overall leniency score is the sum of the three values.

As ours have three level generation parameters and each may take one of three values as described in Sect. III, we exhaustedly generate nine sets of levels by fixing one parameter with a specific value and randomly setting all other parameters each time. To see the controlling effect clearly, we also generate a set of levels by setting all the parameters randomly. Thus, we achieve 10 level sets where each contains 100 levels for reliability. A game level is confined to a 2D map of 200 in width and 15 in height, as same as the setting in previous work, e.g., [5]. In terms of linearity, density and leniency, the expressive ranges of levels controlled by different parameters are shown in Fig. 7 where it is clearly seen that the levels of a specific property are generated by properly controlling a parameter.

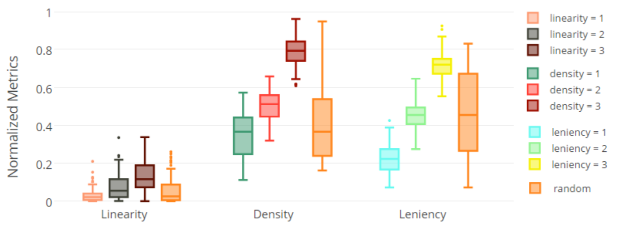

Fig. 7. Expressive ranges of our level generator corresponding to different controllable parameter values.

For exemplification, Fig. 8 illustrates some levels generated by controlling parameters in a specific way. Level in Fig. 8 (C) consists of 14 hills, which leads to highest density value. Levels in Fig. 8 (A) and (B), however, contain 2 and 8 hills respectively, which leads to lower density values. In addition, there are more complicated geometrical structures (e.g., more overlapping hills) in the level shown in Fig. 8 (C) than those shown in Fig. 8 (A) and (B).

Experimental result demonstrates that our generator provides control for designers by controlling relevant content features in CPs locally, and generates a complete procedural level of desired properties.

D. Generation Efficiency

Generation efficiency is often evaluated by the actual time taken in a level generation. By testing on a PC (Intel Core i5-3470 processor with 8GB memory), our level generator takes only 0.057 sec on average to generate a procedural level, 200× 15 2D map, which should be able to meet the online generation requirements.

V. Discussion

In this section, we discuss the issues arising from our work and relate ours to pervious works.

Quality assurance is an open challenge in the area of PCG since automatically generated procedural levels are generally worse than those handcrafted levels [12]. Different types of approaches have been proposed to tackle this issue. For instance, Shaker et al. [6] presented a system which used design grammars and handcrafted fitness functions for generating SMB levels via grammatical evolution algorithm; Smith et al. developed the *Launchpad* generator [7] which used handcrafted critics for quality assurance and designer control. In general, identifying proper constraints and formulating heuristic evaluation functions is difficult since game content is observable but hard to explain and abstract. Thus, Reis et al. [8] merged human-annotated game segments to form complete aesthetical appealing and enjoyable levels. However, they did not train a model to generalize the annotated data. Dahlskog and Togelius [9] used design patterns learned from human-authored SMB levels to generate levels, while Snodgrass and Ontañón [10] learned Markov chains from human-authored SMB levels as constructive rules for level generation. These approaches aim to generate levels with training data (e.g., human-authored levels) instead of domain knowledge. However, infering/learning reliable constructive rules (or diverse design patterns) from these levels without any domain knowledge might be difficult since the number of human-authored levels are limited (e.g., 32 human-authored levels in SMB). In our approach, we explore and exploit the synergy between rule-based and learning-based methodologies to assure the quality of game content. Easy-to-design rules are employed for removal of apparently unappealing game content, while a learning-based approach addresses the rest of quality assurance issues with a single evaluation function. Our data-driven evaluation function implicitly encodes multiple quality-related criteria based on game developers' judgment on quality of training examples, and improves the quality of procedural game with respect to different quality measures. However, a learning-based approach rarely yields the error-free performance, which could be a potential weakness of such approach.

Another issue in PCG is to effectively control the properties of procedural content. In general, a game designer has to encode the desired properties in handcrafted rules (e.g., theory-driven evaluation functions) in order to control the procedural levels (e.g., [5], [7]). While those evaluation functions may work less efficiently especially for generating procedural levels of a considerable length. In contrast, our online level generator clearly benefits from CPs, it provides effective and efficient control for designers. On the one hand, our generator is proposed based on a direct content representation concerning low-level geometrical features. By controlling relevant content features directly, game content of desired properties are directly selected from the specified region of content space. This process is more efficient than aforementioned approaches since using theory-driven evaluation functions to explore the content space is computationally expensive. On the other hand, our representation is working at a local level for CPs. Thus, our generator generates a procedural level efficiently by integrating CPs of desired properties. It is noticed that the desired level properties have to be specified via setting controllable parameters at a local level. This would be a potential weakness when such properties are unknown or hard to specify. In addition, the expressive ranges of procedural levels are also mainly determined by local controllable parameters. Game developer could generate procedural levels of wide expressive range by tuning controllable parameters locally.

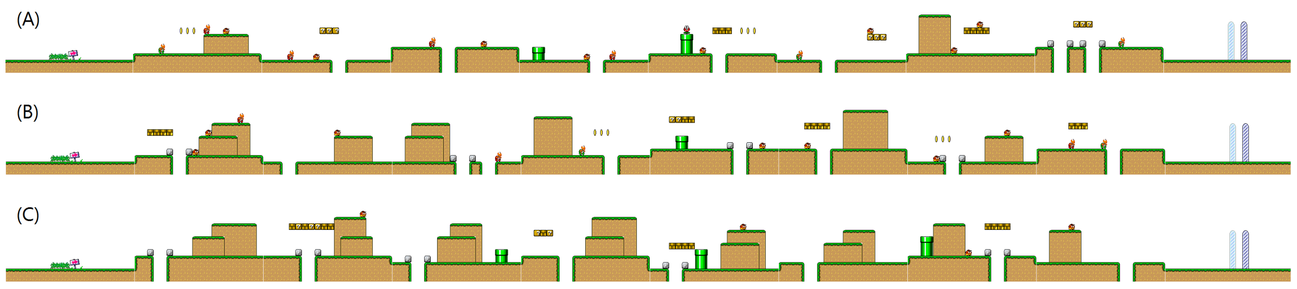

Fig. 8. Exemplar levels generated with different density values. (A) *density* = 1. (B) *density* = 2. (C) *density* = 3.

In general, our online level generator may be viewed as a hybrid PCG approach if we position it in light of the existing taxonomy [1]. On the one hand, we use a generate-and-test method to produce CPs for quality assurance. On the other hand, a procedural level is constructively generated via a number of controllable parameters for effective control. Apparently, ours distinguishes from aforementioned approaches in terms of quality assurance and resultant controllability.

In conclusion, we have presented a novel approach to online level generation in SMB. Our approach can also be used for offline game generation, which allows for using more complex controlling parameters to generate richer content in contrast to our online generation. We explore and exploit the synergy between rule-based and learning-based methodologies to produce controllable yet quality constructive primitives. A complete quality game level can be generated by integrating relevant constructive primitive together via controllable parameters on geometrical features and level properties. We have further carried out a thorough evaluation on our proposed approach and other approaches. The experimental results demonstrate that our approach online generates quality yet controllable levels efficiently. In our ongoing research, we have been working on the application of CPs to generate adaptive games for personalization and extension of this approach to first-person shooter (FPS) games.

Acknowledgment

The authors are grateful to N. Shaker for providing SMB procedural level images used in our comparative study and to A. Liapis who responded to our enquiries for useful discussions.

References

[1] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Trans. Comput. Intell. AI in Games*, vol. 3, no. 3, pp. 172-186, 2011.

[2] Nintendo EAD, "Super Mario World," (Game) 1990.

[3] P. Persson, "Infinite Mario bros," [Online]. Available: http://www.mojang.com/notch/mario/.

[4] N. Shaker, J. Togelius, G. N. Yannakakis, B. Weber, T. Shimizu, T. Hashiyama, N. Sorenson, P. Pasquier, P. Mawhorter, G. Takahashi, G. Smith, and R. Baumgarten, "The 2010 Mario AI championship: Level generation track," *IEEE Trans. Comput. Intell. AI in Games*, vol. 3, no. 4, pp. 332-347, Sep. 2011.

[5] N. Sorenson, P. Pasquier, and S. DiPaola, "A generic approach to challenge modeling for the procedural creation of video game levels," *IEEE Trans. Comput. Intell. AI in Games*, vol. 3, no. 3, pp. 229-244, Sep. 2011.

[6] N. Shaker, M. Nicolau, G. N. Yanakakis, J. Togelius, and M. O'Neill, "Evolving levels for Super Mario Bros using grammatical evolution," in *Proc. IEEE Conf. Comput. Intell. Games*, 2012, pp. 304-311.

[7] G. Smith, J. Whitehead, M. Mateas, M. Treanor, J. March, and M. Cha, "Launchpad: A rhythm-based level generator for 2-d platformers," *IEEE Trans. Comput. Intell. AI in Games*, vol. 3, no. 1, pp. 1-16, Mar. 2011.

[8] W. M. P. Reis, L. H. S. Lelis, and Y. Gal, "Human computation for procedural content generation in platform games," in *Proc. IEEE Conf. Comput. Intell. Games*, 2015, pp. 99-106.

[9] S. Dahlskog and J. Togelius, "A multi-level level generator," in *Proc. IEEE Conf. Comput. Intell. Games*, 2014, pp. 1-8.

[10] S. Snodgrass and S. Ontañón, "Experiments in map generation using markov chains," in *Proc. FDG Workshop on Procedural Content Generation*, 2014.

[11] P. Mawhorter and M. Mateas, "Procedural level generation using occupancy regulated extension," in *Proc. IEEE Conf. Comput. Intell. Games*, 2010, pp. 351-358.

[12] J. Togelius, A. J. Champandard, P. L. Lanzi, M. Mateas, A. Paiva, M. Preuss, and K. O. Stanley, "Procedural content generation: goals, challenges and actionable steps," *Artificial and Comput. Intell. in Games*, vol. 6, pp. 61-75, 2013.

[13] B. Horn, S. Dahlskog, N. Shaker, G. Smith, and J. Togelius, "A comparative evaluation of procedural level generators in Mario AI framework," in *Proc. FDG Workshop on Procedural Content Generation*, 2014.

[14] G. Smith and J. Whitehead, "Analyzing the expressive range of a level generator," in *Proc. Workshop on Procedural Content Generat. Games*, 2010.

[15] C. McGuinness and D. Ashlock, "Decomposing the level generation problem with tiles," in *Proc. IEEE Congr. Evol. Comput.*, 2011, pp. 849C856.

[16] J. Roberts and K. Chen, "Learning-based procedural content generation," *IEEE Trans. Comput. Intell. AI in Games*, vol. 7, no. 1, pp. 88-101, 2015.

[17] S. K. Thompson, *Sampling (second edition).* New York: Wiley, 2002.

[18] Selecting sample sizes [online], http://www.itl.nist.gov/div898/handbook/ppc/section3/ppc333.htm.

[19] S. Guha, R. Rastogi, and K. Shim, "CURE: an efficient clustering algorithm for large databases," *ACM SIGMOD Record*, vol. 27, no. 2, 1998.

[20] J. Han and M. Kamber, *Data mining.* San Francisco, CA, itd: Morgan Kaufmann, 2001.

[21] A. L. N. Fred and A. K. Jain, "Combining multiple clusterings using evidence accumulation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27, no. 6, pp. 835-850, 2005.

[22] C. Chen, A. Liaw, and L. Breiman, "Using random forest to learn imbalanced data," Univ. California, Berkeley, CA, Tech. Rep., 2004.

[23] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5-32, 2001.

[24] J. Togelius, S. Karakowskiy, and R. Baumgarten "The 2009 Mario AI Competition," in *Proc. IEEE Congr. Evol. Comput.*, 2009.

[25] A. Liapis, G. N. Yannakakis, and J. Togelius, "Towards a generic method of evaluating game levels," in *Proc. AAAI Conf. Artif. Intell. Interact. Digit. Entertain.*, 2013.