A procedural procedural level generator generator

Manuel Kerssemakers, Jeppe Tuxen, Julian Togelius and Georgios N. Yannakakis

***Abstract*—Procedural content generation (PCG) is concerned with automatically generating game content, such as levels, rules, textures and items. But could the content generator itself be seen as content, and thus generated automatically? This would be very useful if one wanted to avoid writing a content generator for a new game, or if one wanted to create a content generator that generates an arbitrary amount of content with a particular style or theme. In this paper, we present a procedural procedural level generator generator for Super Mario Bros. It is an interactive evolutionary algorithm that evolves agent-based level generators. The human user makes the aesthetic judgment on what generators to prefer, based on several views of the generated levels including a possibility to play them, and a simulation-based estimate of the playability of the levels. We investigate the characteristics of the generated levels, and to what extent there is similarity or dissimilarity between levels and between generators.**

I. Introduction

Recent years have seen a flurry of interest in creating procedural content generators for different types of game content. These generators build on a large variety underlying techniques, including cellular automata [1], artificial evolution [2], L-systems, answer set programming [3], artificial agents and fractal subdivision. Likewise, generators have been made to generate a large variety of types of content for a similarly large variety of games. Examples include quests and mazes for action adventures [4], complete sets of rules for board games [5] and maps for strategy games.

In almost all cases, a content generator is made to generate a single type of content for a single game. While a paper describing a new generator typically outlines how the PCG algorithm could be brought to bear on other content generation problems, i.e. on other combinations of content type, game constraints and desirable properties, actual and often significant re-engineering is needed to do this in practice. For the mind that is already thinking about how to automatically generate game content, the natural continuation of this line of thought is to think about if not the content generator itself could be procedurally generated.

Smith and Mateas [3] discuss the role of the designer when applying procedural generation techniques to a game. They argue that as the designer takes a step back from designing the actual artefact (e.g. a level), the design effort is spent on shaping the design space of the content generator instead. Of course, if the design of the actual content can be automated, then the design of the design space should also be possible to automate.

The authors are with the IT University of Copenhagen, 2300 Copenhagen, Denmark. (email: m.kerssemakers@gmail.com, {jtux, juto, yannakakis}@itu.dk)

In this paper, we make an attempt to apply PCG techniques to the actual content generator. In other words, we are doing meta-PCG, and removing the designer one step further from the eventual artefact. The main question we address is whether it is indeed possible to procedurally generate content generators. Developing the system that is described in this paper required addressing questions of how to visualise a large range of possible generated artefacts in a compact form, how to ensure playability of levels generated by very unconstrained generators, and how to retain designer control in a two-layered generation process.

II. Background

The work presented in this paper builds on a body of previously published research, dealing with different aspects of creating digital content, especially game content.

A. Super Mario Bros level generation

In the past few years, a number of researchers have addressed the problem of automatically generating playable, entertaining and in some cases even personalised levels for platform games [6]–[8]. The earlier attempts all used different platform games to test the proposed content generators, making comparison of results problematic. To rectify that, the *Mario AI benchmark* was defined based on a modified version of *Infinite Mario Bros*, which is Markus "Notch" Persson's public domain Java-based clone of Nintendo's classic platform game *Super Mario Bros*. Released in 1985, Super Mario Bros redefined the platform game genre and has influenced the design of virtually every platform game since (as well as countless games of other genres); it remains popular to this day. The Mario AI benchmark has been used for a series of competitions focusing on developing AI agents that play the game [9], as well as for a competition focusing on level generation [10]. In the latter competition, competitors submitted a number of very different level generators based on diverse approaches.

Apart from the popularity and design of the platform game that the benchmark software is modelled on, another reason for the appeal of the benchmark and competition seems to be the very simple level representation. A level is simply represented a two-dimensional matrix of height 15 and variable length, typically a few hundred. Each cell in the matrix represents a single "block"; the cell can be empty, or be one of a number of block types, including various forms of impenetrable objects, enemies, collectables and power-ups. The player character, Mario, is about one block wide and two blocks high, and can jump a distance of a few blocks. A level is won by traversing

978-1-4673-1194-6/12/$31.00 ©2012 IEEE

it from the left border of the level to the right border without falling down a gap or getting killed by an enemy.

B. Search-based PCG

Search-based procedural content generation, or SBPCG, refers to the use of evolutionary algorithms of other search/optimisation algorithms to generate game content. A recent survey [2] outlines the major research challenges within this field and lists a number of examples where this approach has been taken to generate content of various kinds. When applying SBPCG to a content generation problem, two key problems are defining the right *evaluation function*, that assigns a number to a candidate content artefact based on its quality or suitability, and defining a good *content representation*. The representation can be very different from the actual content artefact – e.g. a vector of real numbers or rules in a grammar might be interpreted as a weapon or as a personality. The process of creating the content from its representation before evaluating it is called the *genotype-to-phenotype mapping*.

C. Interactive evolution

Finding a good evaluation function (also called fitness function) is a key problem for many applications of evolutionary computation. While some properties of an evolved object are very simple to measure, others are hard to even reliably approximate using an algorithm. Good examples are properties relating to the beauty, interestingness, fun or ingenuity of an artefact. The obvious solution is to include a human as part of the evaluation process: the evolutionary algorithm presents candidate artefacts to a human user, who responds by assigning a fitness value or selecting which ones of the presented artefacts he/she likes best. Interactive evolution, as this approach is called, has been used widely in evolutionary art and music [11], [12].

D. Agent-based PCG

A very different approach to generating game content is to use a large number of independent agents to create the content. These agents might work in parallel or serially, and might or might not have the ability to interact with each other. A good example of this approach is Doran and Parberry's agent-based generation of terrain, where a populations of different agents are let loose on an initially featureless island in different "waves" [13]. One wave creates mountains, followed by another wave of agents that create rivers that flow from the mountains to the sea, followed by shoreline agents that create nice beaches along the coast, followed by erosion agents etc. While the results look good, it is rather hard to control that gameplay-critical constraints on the terrain (e.g. the existence of traversable routes out of valleys) are observed.

E. Combining content generators

In the aforementioned SBPCG survey, it is observed that the genotype-to-phenotype mapping can be considered a procedural content generator in its own right [2]. This provides an opportunity to combine content generators, by putting an "inner" PCG algorithm as the genotype-to-phenotype mapping within a search-based "outer" PCG algorithm. What is evolved in the outer generator is in other words the parameters of the inner generator. In a recent paper, this strategy is discussed and it is argued by using the inner generator to ensure well-formedness and playability, the outer generator can focus on ensuring the less tangible aesthetic properties, such as interestingness and fun; this way, the compositional content generator can combine the best properties of both its constituent algorithms [14]. This is demonstrated by generating dungeons for a fictive Roguelike game using answer set programming as an inner generator for guaranteeing basic playability, and an evolution strategy as an outer generator with an evaluation function designed to measure the challenge of the dungeon.

III. System overview

The system presented in this paper can be seen as a compositional procedural content generator, with an interactive genetic algorithm as its outer generator and an agent-based algorithm as the inner generator. The domain is levels for Infinite Mario Bros, using the Mario AI benchmark; the inner generator guarantees that levels are playable using simulation-based testing, and the in outer loop the user decides which inner generator generates the most interesting levels. As the agent-based procedural content generator is parameterised using a rather flexible language, it can also be seen as an attempt to evolve the inner generator itself, with the outer generator becoming a procedural procedural level generator generator.

A. Interface and workflow

Figure 1 describes the workflow of the system, seen from the perspective of a user using it as a tool via the graphical user interface. The tool starts by selecting 12 generators at random from a database of viable (inner) generators. This database will be described further in the section about the genetic algorithm (GA). These 12 generators make up generation 0 and are visualised to the user. Each generator can be visualised in several different ways. To begin with, a "cloud map", which can be briefly described as the average level the generator output, is shown for each generator. The cloud map is created by generating a number of maps and at each tile position averaging the legend colour of the there present tile in each map. The second visualisation is a playability rating which shows to what extent the levels a generator creates are playable by the AI. A user can right-click to further inspect a generator and is then shown a sample of its generated levels. On clicking one of these levels it can be played in the Mario interface to verify the expected gameplay value. Now, as long as the user is not satisfied by any of the generators, they can iterate through the GA. This is possible by clicking generators with preferred features to mark them as parents and then clicking a button to go to the next generation. When a satisfying generator is found, its levels can be played endlessly. Right now this is only possible from within the tool. Created generators can be exported to be fed into new runs of the algorithm.

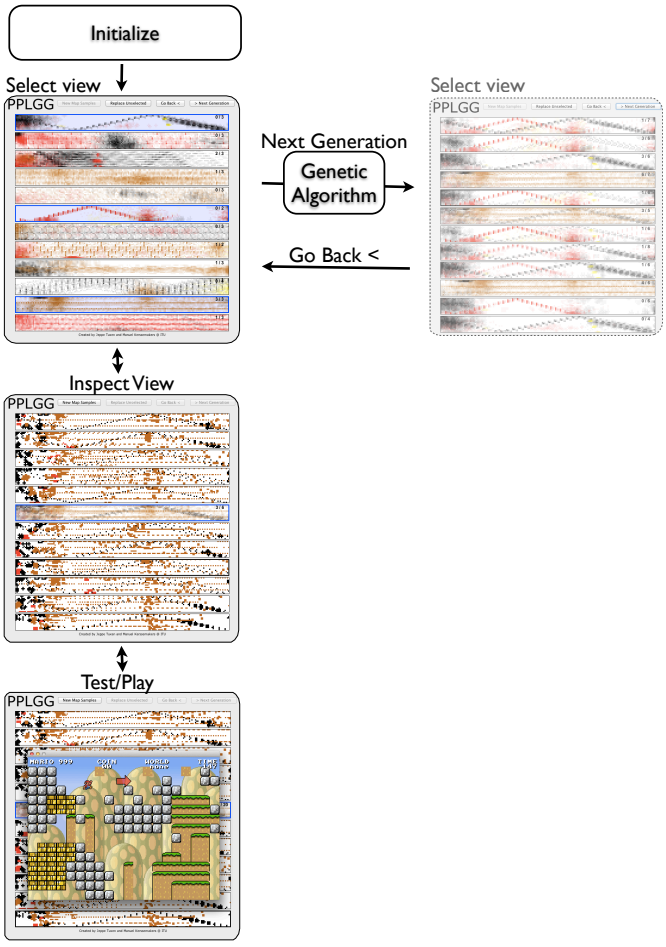

Fig. 1. The workflow of the PPLGG GUI.

IV. Inner Generator

The inner generator is defined as a collection of agents. Each agent is specified with a set of attributes and behaviours, much like the software agents in Doran and Parberry's work [13]. Our approach differs from theirs in several ways, most importantly in that while Doran and Parberry uses hard-coded agents with specialised behaviours like (like "coastline agents" and "smoothing agents"), we use one general agent – the GAgent – that can be parameterised to provide a large variety of behaviours. The reason for choosing the GAgent over hard-coded is to keep the space of possible content generators as large as possible. The results of this approach is indeed that a large variety of different generative behaviours can be specified, most of which would be unlikely to contribute to building entertaining levels. It is therefore up to the fitness function in the GA to decide whether or not a randomly initiated GAgent will survive as part of a generator.

A generator contains anywhere between 14 an 24 agents (these numbers were determined experimentally to be suitable for the chosen level length), which modify the level through moving around on it and "drawing" or "erasing" blocks of different kinds. The agents move independently and usually concurrently. They can not communicate with each other except through modifying the level.

An agent is defined by a number of parameters, that specify how it moves, for how long, where and when it starts, how it changes the level and in response to what. The agent's behaviour is not deterministic, meaning that any collection of agents (or even any single agent) is a level generator that can produce a vast number of different levels rather than just a generative recipe for a single level.

The first five parameters below are simple numeric parameters that consist in an integer value in the range specified below. The last five parameters are categorical parameters specifying the logic of the agent, which might be associated with further parameters depending on the choice of logic.

- **Spawn time [0-200]**: The step number on which this agent is put into the level. This is an interesting value as it allows the sequencing of agents, but still allows for overlap.
- **Period [1-5]**: An agent only performs movement if its lifetime in steps is divisible by the period.
- **Tokens [10-80]**: The amount of resources available to the agent. One token roughly equals a change to one tile.
- **Position [Anywhere within the level]**: The center of the spawning circle in which the agent spawns.
- **Spawn radius [0-60]**: The radius of the spawning circle in which the agent spawns.
- **Move style**: the way the agent moves every step.
  - follow a line in a specified direction (of 8 possible directions) with a specified step size.
  - take a step in a random direction
- **Trigger state**: The condition for triggering an action, checked after each movement step.
  - always
  - when the agent hits a specified type of terrain.
  - when a specified rectangular area is full of a specified tile type
  - when a specified area does not contain a specified tile type
  - with a specified probability
  - with a specified time interval
- **Boundary movement**: The way the agent handles hitting a boundary.
  - bounce away
  - go back to start position
  - go back to within a specified rectangular area around the start position
- **Action type**: The type of action performed if it is triggered.
  - place a specified tile at position
  - place a rectangular outline of specified tiles and size around position
  - place a filled rectangle of specified tiles and size around position
  - place a circle of specified tiles and size around position

– place a platform/line of specified tiles and size at position
– place a cross of specified tiles and size at position
– place or remove specified tiles in a specified area according to the rules of Conway's "Game of Life" [15].

The core loop of the agent is very simple, as almost all of the intelligence depends on the logic specified by the parameters above. Every tick, each active agent first moves, then checks whether it has to react to level boundaries. Then it checks whether its action is triggered and if so, it performs its action. Each time an action is performed, the agent consumes a token; it continues until it has run out of tokens.

V. Outer generator

The inner generators are parameterised (or, depending on your perspective, generated) by the outer generator. As described below, this is an interactive genetic algorithm where the user makes the ultimate quality judgment but is aided by simulated play-through for quality checking.

A. Interactive genetic algorithm

The genetic algorithm keeps a population of 10 individuals. Each individual is a level generator, and is simply represented as a list of its constituent agents. The agents, in turn, are represented by the values of all their parameters, as described above. Each generation, it displays all 12 individuals to the user, and the user chooses one or more individuals to form the basis of the next generation. When the user chooses to move on to the next generation, an entirely new population is generated based on mutation and crossover of the selected parents.

As mentioned above, the users are aided in their decisions about which levels to select as parents by being able to experience them in several forms: as a cloud visualisation, as visualisations of individual generated levels, through playing individual levels, and through a playability rating. Figure 2 shows a level as represented in the individual level view, and how a small slice of it looks in the game view. The playability rating is calculated automatically by the program through sampling ten levels from each generator, and letting an AI try to play them to completion. In this version of the generator, we use Robin Baumgarten's Mario-playing agent, which won the 2009 Mario AI competition and is based on searching the state space of the game with A*. The playability ratio is simply defined as the average proportion of the ten levels that the agent manages to clear before dying. It is perfectly possible, and perhaps desirable (though this is up to the user), for a level generator to have a playability rating of 1.

Initial testing revealed a problem in that a large proportion of completely randomly initialised generators only generate levels that are neither playable nor interesting. Furthermore, in a population of 10 randomly initialised level generators some agent types might be missing altogether, for example there might not be any agents that add coins or there might be no agents that draw platforms to stand on in the second half of the level. This means that there might not be enough good genetic material in the first generation to get evolution started, as long as evolutionary runs are completely randomly seeded.

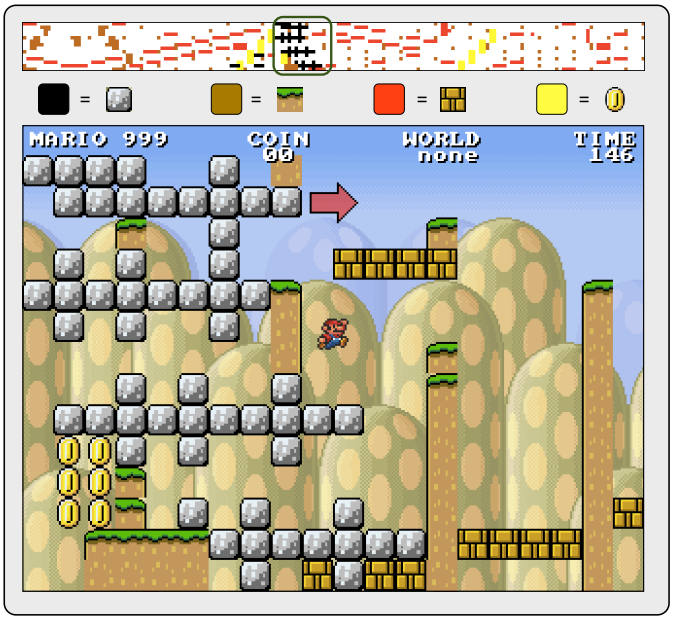

Fig. 2. A single generated level, and a small part of the same level in the game view.

We therefore introduced an initialisation step, to ensure a reasonably high playability ratio for the initial generators. Unlike in the interactive GA this process of finding suitable candidates for the database is done offline and is non-interactive. To filter out a suitable generator the GA evaluates fitness of a randomly seeded generator by playing three of its levels and then returning the average traversed horizontal distance among the play-throughs, representing playability. After the evaluation step it applies tournament selection to determine parents and uses the mutation (50%) and crossover (90%) operators to produce a new population. The algorithm returns a level generator whenever one is found that produces only playable levels, or after a specified number of generations (100). This generator is then added to the database selectable for use as a starting point for the interactive GA. Figure 3 shows the generators that were created in the initialisation step of an example run.

B. Mutation and crossover

Crossover of two generators is defined as follows. A random number between 0 and 1, r, is determined. then $r * size\_of\_parent1$ agents are picked randomly from parent 1, and $(1 - r) * size\_of\_parent2$ agents are picked at randomly from parent 2. Together these agents make up the new generator. No cross-over is done between agents themselves, since generators need to swap features and features are mainly recognised as entire agents.

Mutation changes one of the agents by shifting all attributes by a small amount. Mutation of a generator can also add a new

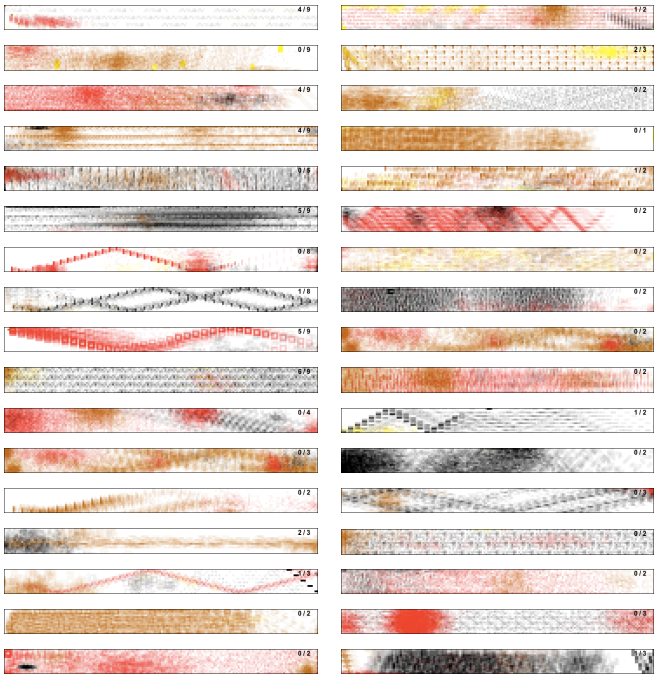

Fig. 3. The database of initial content generators (cloud view).

agent or remove a randomly chosen agent. The results of the crossover and mutation operators are visualised in figure 4

VI. Evaluation

This section reports on informal self-evaluation of the system and associated tool by the designers.

A. User experience

Searching a large space by stochastic methods usually involves a very large number of evaluations. This is a problem for interactive evolution, as human evaluation of candidate artefacts takes time and effort, and humans have a limited attention span. The problem of users losing interest in partaking in the evolutionary process after a number of generations is called *user fatigue*. In the current system we have tried to circumvent user fatigue with the initialisation process, which ensures a high playability rate of every generator saved in the database. This way users can start working with their preferences right away, instead of having to select for playability.

Based on our own experience of the system, we consider this approach to have succeeded given the broad design goal that it should be easy for a user to generate playable and interesting levels. However, for a user looking to create a generator that creates some specific level feature, it can still take many generations before that goal is achieved, if ever.

Most importantly, we felt in control in the selection process. This is partly because of its high level of transparency: when comparing parents and their off springs it is very clear from which parent each feature comes from.

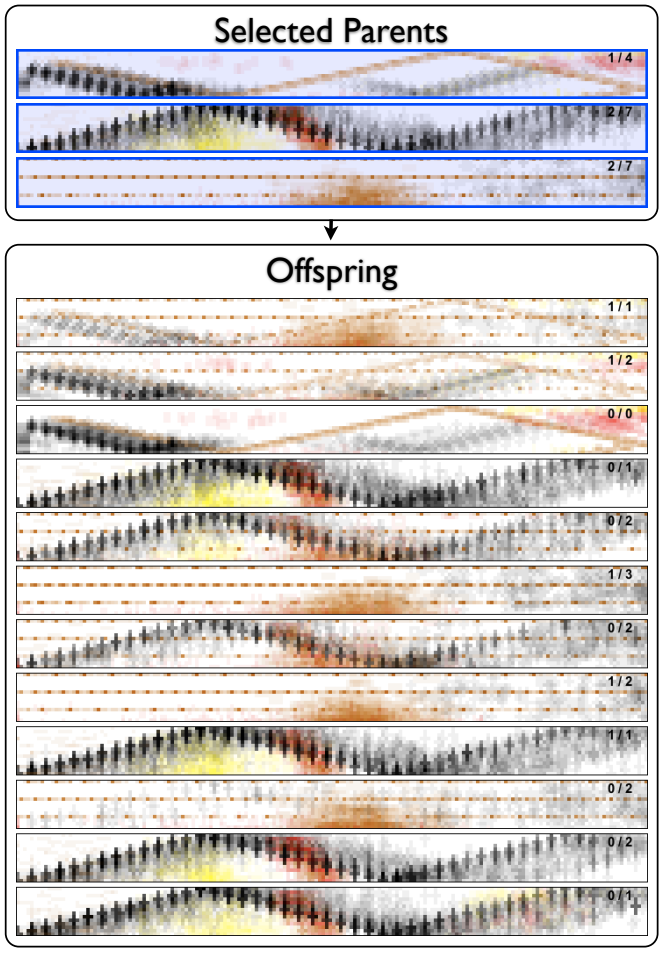

Fig. 4. Offspring, generated from three parents by recombination and mutation. (All generators shown in cloud view.)

B. Diversity of generated levels

The amount of variability between levels differs from generator to generator: some generators seem to produce essentially the same level with minor variations, others generate levels that look completely different from each other in the game view, while retaining a common "theme" or "style". (From the higher-level inspector view it is almost always easy to see which levels are created by the same generator.) This variance is mostly due to the variance in the agent spawn radius. This attribute states in which radius from its starting position an agent may be spawned. A high value means that levels from that generator will be more different locally. Figure 5 shows an example of the same area in four different levels from the same generator, and discusses their similarities.

Within the same level, one can likewise often see large differences between different parts. This is due to that some agents are only active within part of a level, depending on the starting position, number of tokens and movement logic. Different levels generated by the same generator usually show the same differences between parts; for example, one generator might generate levels that all have many more coins in the

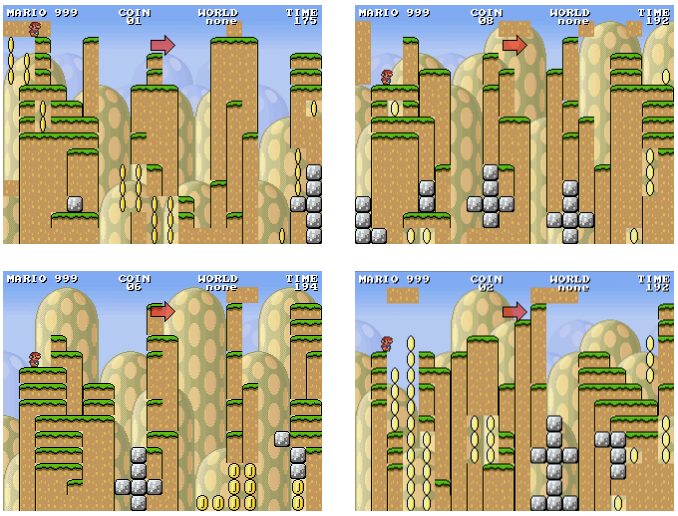

Fig. 5. The start of four levels (the first screen) generated by the same generator. One can immediately see that the four level segments are related, so to speak created in a particular style. Certain higher-level descriptions hold true for all four levels, e.g. that there are multiple short platforms stacked on top of each other and vertically elongated fields of coins. A theme that seems to recur is solid stone blocks in cross-shaped patterns. At the same time, the segments are completely different from each other: no individual features can be found in the same place in the four levels.

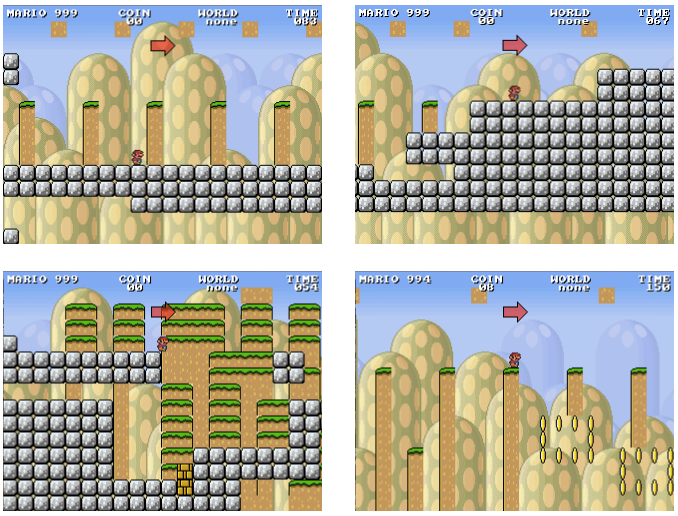

Fig. 6. Typical variance within a level. Four screens from different parts of the same level. No clear common theme can be discerned, apart from the absence of enemies and the near-absence of brick-blocks.

second half than the first, even though they differ in the precise locations of these. Figure 6 shows four screens taken from different parts of the same level.

C. Performance

For real-time level generation, or for use as a genotype-to-phenotype mapping in artificial evolution, a content generator needs to be blazingly fast. To measure the speed of an arbitrary procedural level generator we sampled 100 levels each from 100 random generators. To test the scalability of the algorithm over different map sizes, we tested with three different several widths (the "standard" width is 180 blocks, corresponding to levels that take 1-2 minutes to play). As randomly initialised level generators tend to produce unplayable levels, we also redid the test with generators randomly sampled from the database of diverse level generators used for initialising evolutionary runs. From the results in table I we can see that generation time increases approximately linearly with level length, and that the generators from the database are significantly slower than randomly created generators. The latter is probably due to that playable levels are characterised by being more dense than the unplayable levels generated by many random generators. The most important fact, however, is that levels of ordinary length (300 blocks) can indeed be generated blazingly fast – more than 20 per second on the standard laptop we used for testing. This opens up for using the generators online, for real-time level generation in response to user actions. One could imagine a game where each level is generated by a different generator. Each of these would have different features and difficult, thus making the game different on each play through, but still retaining a reasonable learning curve and a set narrative progression and thematic diversity throughout the length of the game.

width	time
80	5ms
180	16ms
1000	84ms

width	time
80	24ms
180	45ms
1000	160ms

TABLE I
AVERAGE GENERATION SPEED PER LEVEL RELATIVE TO LEVEL WIDTH FOR RANDOM GENERATORS (LEFT) AND DATABASE GENERATORS (RIGHT).

VII. Discussion

Originally, the procedural procedural level generator generator described in this paper was meant to be able to generate levels for any games for which levels could be described as a two-dimensional matrix (including other platformers, Roguelikes, Zelda-style action-adventures etc). The constraints of the particular game would be included as parameters for the generators. During work with the system, this turned out to be a too ambitious goal at this stage. However, in principle it would be possible to create a procedural generator of procedural level generators that is to a large extent game-agnostic. When prototyping a new game, it would be no doubt be very useful to be able automatically generate interesting levels for it. Key stepping stones towards making this system more generic would be to develop a language in which to encode the playability constraints of the new game, and a way of automatically learning an AI that can play the new game proficiently in order to test the playability of the levels.

In the background, we discussed a compositional procedural content generator which uses an evolution strategy as an outer generator and an inner generator based on answer set programming (ASP) [14]. It is interesting to compare that system with the system described in the current paper: both can be seen as compositional content generators, or as procedural procedural content generator generators. In both cases, the

inner generator could at any point be removed from the outs generator and produce a huge number of unique levels. Apart from the domain and type of evaluation function, the main difference is that in the aforementioned paper the inner generator was expressed in ASP and in the current system it is expressed as logics and parameters for a population of agents. The strength of ASP as an inner generator is that it can guarantee that certain constraints are met, in particular such as having to do with well-formedness and playability. The agent-based approach cannot guarantee the satisfaction of any constraints at all – this is why the simulation-based testing using an AI player is needed. However, the agent-based generator has the distinction of producing levels with a certain style. Cursory inspection of randomly chosen outputs from any given ASP-based dungeon generator show no obvious commonality in style; the agent-based generators, on the other hand, do.

Another aspect of the generators generated by the current system that should not be overlooked is that many of them generate levels that quite simply do not look like any platform game levels we have seen before. Most of the generators submitted to the Mario AI level generation competition [10] rely on placing (possibly parameterised) pre-specified elements of different kinds. Other generators, like Tanagra, use completely different methods for generating level geometry but have other limitations (e.g. the current Tanagra implementation seems never to generate branching paths or vertical stacking of any kind) [7]. At their best, the levels generated by the agent-based generators manage to break conventions about how certain blocks should be placed relative to each other, while still seeming purposeful. For those who believe that one of the reasons for engaging in PCG research is to allow for machine creativity beyond the constraints humans seems to apply to themselves, this should be good news.

VIII. Conclusion

We have presented a procedural procedural level generator generator, where an interactive evolutionary algorithm evolves agent-based generators consisting of populations of agents that draw and erase blocks whilst traversing the level canvas. Alternatively, the system can be understood as a compositional procedural content generator with an interactive evolutionary algorithm as its outer generator and an agent-based inner generator. The two perspectives are complementary rather than exclusive. We believe the system could relatively easily be generalised to games of other genres, given that the game rules could be expressed in an appropriate language and an AI to play the game be learned.

Acknowledgments

The research was supported in part by the Danish Research Agency (FTP) project *AGameComIn* (number 274-09-0083).

References

[1] L. Johnson, G. N. Yannakakis, and J. Togelius, "Cellular Automata for Real-time Generation of Infinite Cave Levels," in *Proceedings of the ACM Foundations of Digital Games*. ACM Press, June 2010.

[2] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: a taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, pp. 172–186, 2011.

[3] A. Smith and M. Mateas, "Answer set programming for procedural content generation: A design space approach," *IEEE Transactions on Computational Intelligence and AI in Games*, 2011.

[4] J. Dormans, "Engineering emergence," Ph.D. dissertation, Amsterdam University, 2012.

[5] C. Browne, "Automatic generation and evaluation of recombination games," Ph.D. dissertation, Queensland University of Technology, 2008.

[6] K. Compton and M. Mateas, "Procedural level design for platform games," 2006.

[7] G. Smith, J. Whitehead, and M. Mateas, "Tanagra: Reactive planning and constraint solving for mixed-initiative level design," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 201–215, 2011.

[8] C. Pedersen, J. Togelius, and G. N. Yannakakis, "Modeling Player Experience for Content Creation," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 1, pp. 54–67, 2010.

[9] S. Karakovskiy and J. Togelius, "The mario ai benchmark and competitions," *IEEE Transactions on Computational Intelligence and AI in Games*, 2012.

[10] 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 Transactions on Computational Intelligence and Games*, 2011.

[11] H. Takagi, "Interactive evolutionary computation: Fusion of the capacities of EC optimization and human evaluation," *Proceedings of the IEEE*, vol. 89, no. 9, pp. 1275–1296, 2001.

[12] J. Romero and P. Machado, *The art of artificial evolution: a handbook on evolutionary art and music*. Springer-Verlag New York Inc, 2007.

[13] J. Doran and I. Parberry, "Controllable procedural terrain generation using software agents," *IEEE Transactions on Computational Intelligence and AI in Games*, 2010.

[14] J. Togelius, T. Justinussen, and A. Hartzen, "Compositional procedural content generation," in *Proceedings of the FDG Workshop on Procedural Content Generation*, 2012.

[15] M. Gardner, "Mathematical games: The fantastic combinations of john conways new solitaire game life," *Scientific American*, vol. 223, no. 4, pp. 120–123, 1970.