Constraint-Based 2D Tile Game Blending in the Sturgeon System

Seth Cooper[1]

[1]Northeastern University, Boston, MA, USA

Abstract
In this work, we explore various approaches to blending multiple games, by building on the Sturgeon constraint-based 2D tile level generation system. We introduce a *game grid*, which, when used as an input to the generator, allows elements such as tiles, tile patterns, and reachability rules to be specified on a per-location basis. We demonstrate how the system can: use the layout of one game's levels for another game's tile patterns; generate levels with one game's tile patterns that use another game's reachability rules; and generate levels that combine tiles, tile patterns, and reachability rules from multiple games.

Keywords
procedural level generation, constraint solving, game blending

1. Background

This work explores various applications of blending different games to generate levels using the Sturgeon constraint-based level generation system [1]. We introduce a *game grid*, that allows various properties of the generated level to be specified on a per-location basis. These properties include tiles, tile patterns, and reachability rules. We demonstrate several applications of blending across multiple games from different genres, including blending layouts and tile patterns, blending reachability and tile patterns, and blending multiple games into a single level.

Blending different games can be seen as *combinational creativity*, whereby new ideas can be produced by combining familiar ones [2]. This approach might lead to new game ideas or inspirations.

This work falls in the domain of procedural content generation (PCG) [3]. It incorporates tile patterns learned from example levels, similar to WaveFunctionCollapse [4] and model synthesis [5], and therefore can be considered a form of procedural content generation via machine learning (PCGML) [6].

Prior work in PCG has used constraint-based level generation, including work using Answer Set Programming [7, 8] and constrained MdMC [9].

Work has also been done in the area of blending games or levels; for example, using Variational Autoencoders to enable controlled blending between platform game levels [10] or to generate blends of dungeon and platformer levels [11]. Some work has proposed the combination of different machine-learned models for content generation [12], such as blending models of levels learned from gameplay videos [13]. Other work has examined blending parts of multiple games into a single level, guided by "sketches" and partitioning [14]. Platformer jump physics have been extracted from blended levels [15].

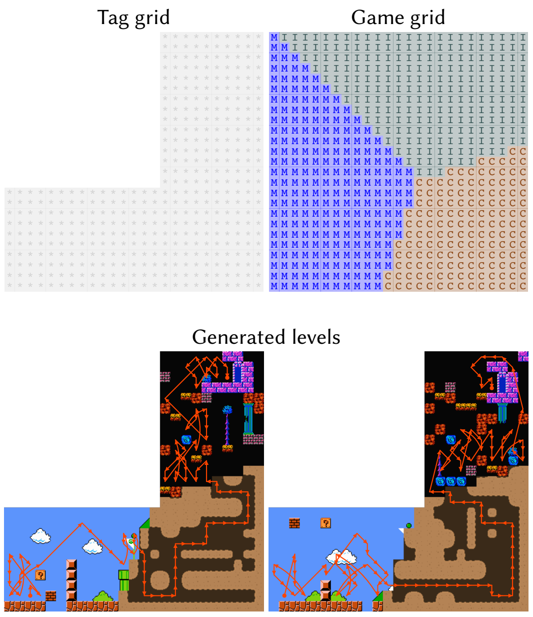

Figure 1: A level generation example blending three games — `mariobros` (M), `cave` (C), and `icarus` (I) — into a single level. The top row shows the input *tag* grid, which specifies where and which tiles can be placed, and the *game* grid, which specifies which game's information can be used at each location. The bottom row shows generated levels, with the red line showing the reachability path through the level (notably, paths can be circuitous). How the player can move changes within the level from game to game. Median, maximum generation times were 3s, 6s.

The Joint Workshop Proceedings of the 2022 Conference on Artificial Intelligence and Interactive Digital Entertainment
se.cooper@northeastern.edu (S. Cooper)

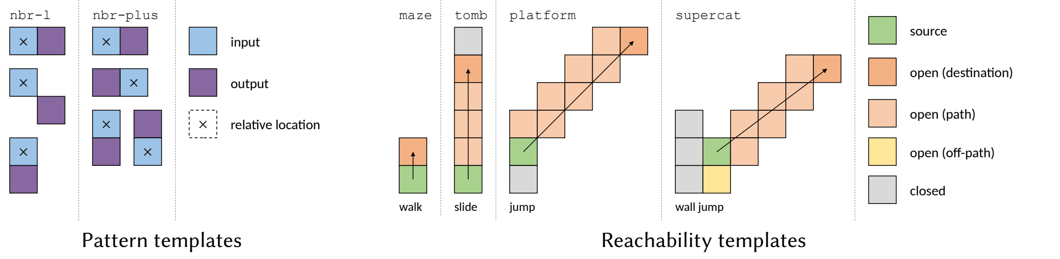

Figure 2: Pattern and reachability templates used in this work. Pattern templates are used to learn and apply tile patterns. Reachability templates are used to ensure the goal of the level is reachable form the start; full reachability templates are not shown, only one example is given for each.

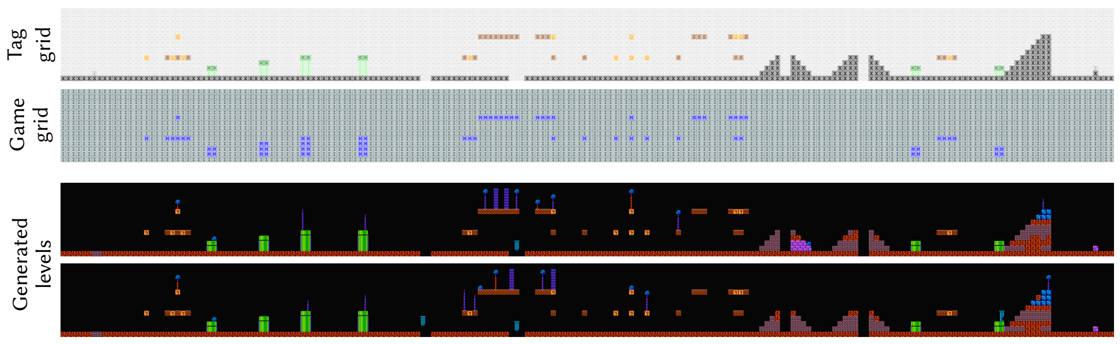

Figure 3: `mariobros` (**M**) level used as functional layout to generate a mainly `icarus` (**I**) level. Median, maximum generation times were 7s, 7s.

Recent work has explored style transfer between games [16], similar to the layout and image blending presented here.

2. Approach

2.1. Sturgeon

This work is an extension of the Sturgeon system. We give a brief overview of the system here to clarify some terms used in the applications. A more thorough description is given in [1].

Sturgeon works on 2D tile-based games. A tile is an entity that can be placed at a location in a grid; tiles can have associated with them either an image (e.g. brick), functional information (e.g. solid), or both. A level is thus a grid of tiles. The original Sturgeon also works with *tag* grids, where a tag has some number of tiles associated with it. A tag grid can be provided with an example level to associate tiles with tags, and can be used to specify what tiles can be placed at a location in a generated level. Levels in Sturgeon are considered to exist inside an infinite grid of *void* tiles and tags.

Sturgeon generates levels via solving constraint problems. Level design rules are specified using a high-level solver API that allows constraints of counts and implications over Boolean variables. These constraints are then translated to a low-level solver for efficient solving.

As one type of level design rule, Sturgeon can extract tile patterns from an example level and tag grid. These patterns specify local constraints between tiles in generated levels (e.g. a top-right pipe tile must be to the right of a top-left pipe). Sturgeon supports different *pattern templates* which specify how these relationships are extracted from example levels and applied (via constraints) to generated levels. Pattern templates are shown in Figure 2(left).

As another rule, the distribution (and if desired, general locations) of tiles in generated levels can be constrained to be similar to that of the example level.

Additionally, Sturgeon provides reachability rules, where a level must have a start and goal and the goal must be reachable from the start. This is supported by an additional set of constraints on finding a path through a reachability graph. Roughly speaking, the nodes in the graph correspond to locations in the grid, and edges represent which nodes are potentially reachable from other nodes. Whether a node is actually reachable or not via an edge depends on the placement of *open* (i.e. traversable by the player) and *closed* tiles. How a player

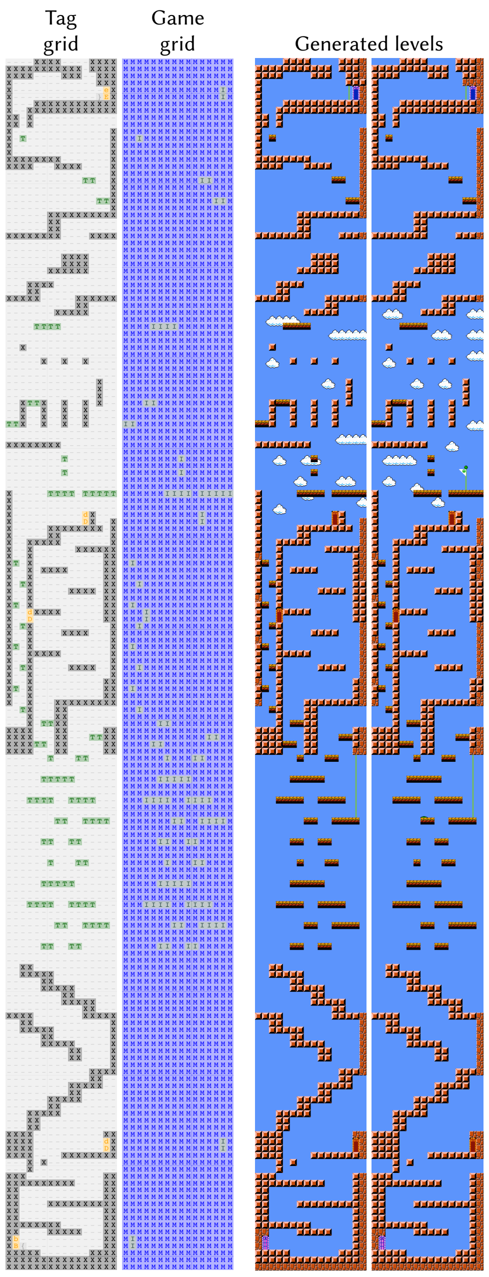

Figure 4: `icarus` (I) level used as functional layout to generate a mainly `mariobros` (M) level. Median, maximum generation times were 26s, 27s.

can move in a game is represented by a reachability template, which is a discrete, tile-based approximation of the player's (potentially continuous) movement. One interesting property of this approach is that it only requires there to be a path, but it does not have to be a short path,

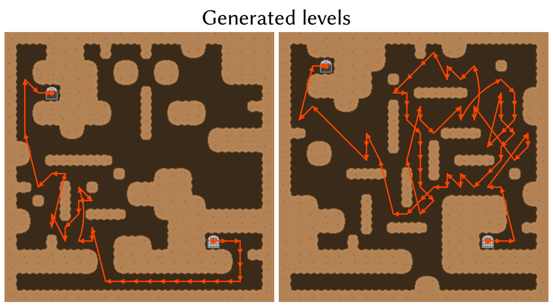

Figure 5: Patterns from `cave` with `platform` reachability. Median, maximum generation times were 1.3m, 2.9m.

and thus often circuitous ones are found, even though shorter ones exist. Some selected reachability templates are shown in Figure 2(right).

2.2. Game grid and extensions

Here we discuss some extensions and modifications to Sturgeon to support multi-game blending.

The main extension to Sturgeon in this work is support for *game* grids. Similar to tag grids, game grids can be supplied with examples to attach information to a particular game; if no game grid is provided, a *default* game grid is used. While a tag has tiles associated with it, a game can have patterns, movement rules, tags, and tiles associated with it. When a game grid is provided along with an example level (in this work, containing only a single game), the information extracted from the level is associated with that game. When a game grid is provided to the generator, the generator uses the information associated with the game specified at each location.

Notably, a game is defined at every location in the grid (even off in the infinite void). This approach was used since in Sturgeon, patterns can start in locations with void tags and constrain tiles in non-void tags, and thus even locations with void tags need a game associated with them.

Other bookkeeping changes were made to Sturgeon to support game blending. The input and data processing routines were updated to handle multiple example levels at once. Also, the pattern *placeability* determination (i.e. whether it is possible to actually place a pattern at a particular location) was changed to be based only on the tiles available at each location, rather than the tiles and tags.

In this work we chose to use the smaller pattern templates `nbr-l` and `nbr-plus`, shown in Figure 2(left). We found these still maintained tile neighbors, but gave the generator flexibility with tile placement. This flexibility helped when, for example, generating levels for a differ-

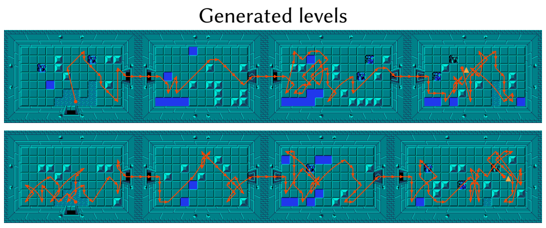

Figure 6: `zelda` patterns with `platform` reachability. Median, maximum generation times were 14s, 16s.

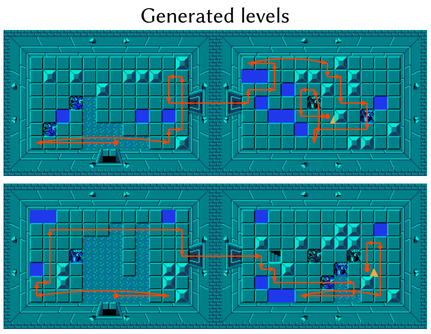

Figure 7: `zelda` patterns with `tomb` reachability. Median, maximum generation times were 16.2m, 32.1m.

ent game's reachability. Example reachability templates are shown in Figure 2(right). Also, in applications where there is more that one game in the game grid, we made the pattern constraints *soft*, so that the generator could violate them if needed, but would try to minimize the number of such violations.

Although Sturgeon has support for multi-pass level generation, in this work we use *simultaneous* generation, where each tile has both an image and a function, and the image and functional grid for a level is created in one step. Additionally, while Sturgeon supports multiple low-level solvers, for this work we use PySAT's [17] RC2 solver [18] with MiniCard cardinality constraints [19], as that was found to be effective in previous work.

2.3. Games

Here we describe the games, along with the pattern and reachability templates used in this work. These games are essentially a subset of those used in [1].

• `cave`: A simple custom cave game. Uses a custom example level with sprites used from Kenney [20] and the `nbr-plus` tile pattern template. Uses the `maze` reachability template, which is single tile four-direction movement.

• `mariobros`: Super Mario Bros [21]. Uses level 1-1 from the VGLC [22], with minor cleanup, as an example and

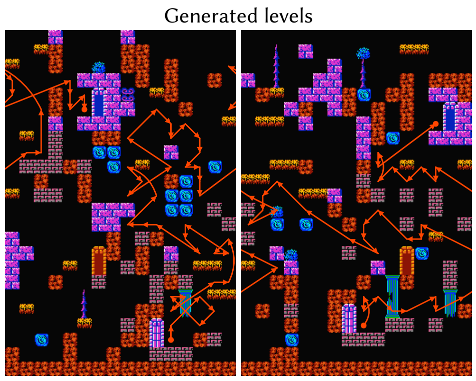

Figure 8: `icarus` patterns with `supercat` reachability and column-wrapping. Median, maximum generation times were 3.8m, 7.2m.

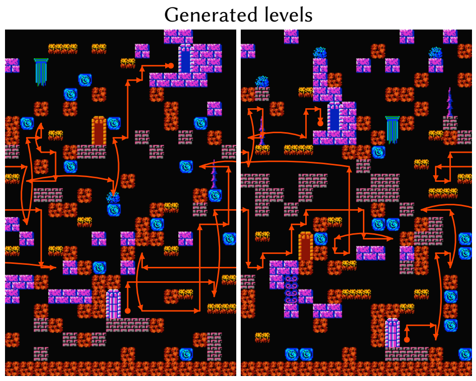

Figure 9: `icarus` patterns with `tomb` reachability and column-wrapping. Median, maximum generation times were 32s, 2.5m.

the `nbr-l` tile pattern template. Uses the `platform` reachability template, which includes platform-style movement such as walking, falling, and jumping, inspired by the pathfinding in [23].

• `icarus`: Kid Icarus [24]. Uses level 1 from the VGLC [22], with minor cleanup, as an example and the `nbr-l` tile pattern template. Uses the `platform` reachability template. Some blends involving `icarus` also allow column-wrapping.

• `supercat`: Super Cat Tales [25]. Uses the tree section of level 1-7 as an example and the `nbr-l` tile pattern template. Uses the `supercat` reachability template, which is a simplification of the game's continuous platform-style movement, in which the player can run up walls, but only jump off walls and ledges.

• `tomb`: Tomb of the Mask [26]. Uses level 1 as an example and the `nbr-plus` tile pattern template. Uses `tomb` reachability template, in which the player can move four

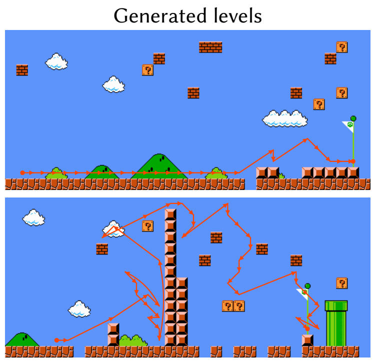

Figure 10: `mariobros` patterns with `supercat` reachability. Median, maximum generation times were 4s, 13s.

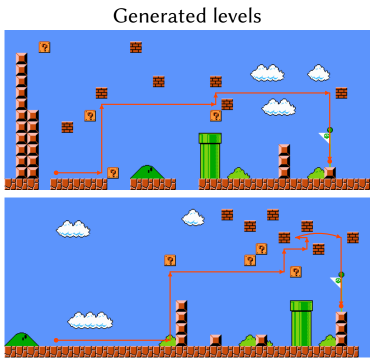

Figure 11: `mariobros` patterns with `tomb` reachability. Median, maximum generation times were 8s, 15s.

directions and slides until hitting something.

• `zelda`: The Legend of Zelda [27]. Uses dungeon 1-1 from the VGLC [22], with minor cleanup, as an example and the `nbr-1` tile pattern template. Uses the `maze` reachability template.

3. Applications

Here we describe some applications of multi-game blending. For each example in the applications, we generated 10 levels for timing information. Levels were generated on a 2018 MacBook Pro. Levels shown in figures were hand-selected from these to indicate the variety of levels generated as well as highlight potential artifacts in the generated levels.

3.1. Functional Layout–Image Blending

In this application, we generate a level for one game that follows the functional layout of another game's level. We use levels from both games as example levels (in this case, `mariobros` and `icarus`). Then we use the functional grid from one game as the tag grid, and the game grid specifies the other game. However, instead of using a uniform game grid, wherever there is a tag that is not known by the other game, the tag grid's game is used at that location to allow its tiles to be placed instead. This results in a level that effectively uses the functional layout of one game with the images from another. This approach does rely on the games sharing common tags. In these examples we did not apply reachability rules, and we used 5x5 regions for the distribution rules to try to capture some tile location preferences.

Figure 3 shows the `mariobros` level used as a tag grid to generate an `icarus` level. Notably, the generator does not place either `mariobros`'s flagpole or `icarus`'s door, which might make the goal unclear. The distribution regions also place blue blocks only at higher locations. Since the two games do not have the same background color, the pipes still have a bit of blue sky behind them.

Figure 4 shows the `icarus` level used as a tag grid to generate a `mariobros` level. Notably, clouds appear partway up the level, as captured by the distribution regions. Also, in the top-right near the exit, the generator is able to place the exit door, and has placed part of a flagpole next to the door. The generator is also able to place bits of the flagpole where the bottom or top connect to the `icarus` game.

3.2. Tile Pattern–Reachability Blending

In this application, we generate a level using tile patterns from one game and the reachability template from another game, essentially generating a level from the first game that can be played by the second.

To do this, we simply specify the reachability template from a different game to be used when generating a level. Each game's level was used as an example independently. Special game grids were not necessary.

Figures 5–11 show levels generated using patterns from one game and reachability from another. Interestingly, when using `supercat` reachability, the generator is able to place gaps and walls that can be used to jump. When using `tomb` reachability, the generator has to place blocks so that the player can stop sliding.

For `zelda` patterns with `tomb` reachability (Figure 6), we had to use soft pattern constraints. This appears to be due to the goal tile being the top-left of the Triforce, which was surrounded by open tiles in the example level, thereby preventing a block being placed next to it for the player to stop on the goal. This can result in an incomplete Triforce being generated. The use of solid stairs also allowed the `tomb` reachability to stop at them. These levels also took notably longer (up to half an hour) to generate.

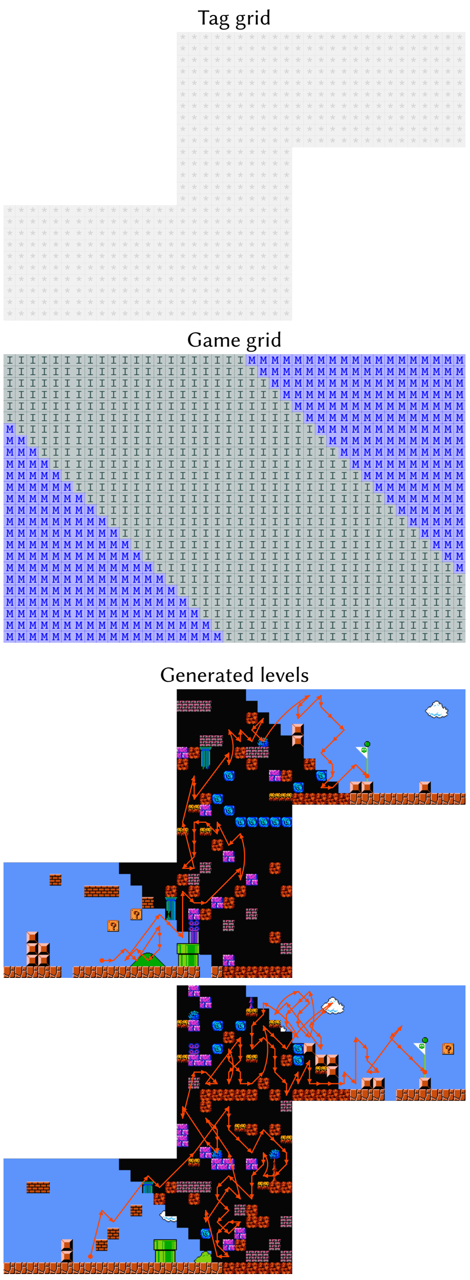

Figure 12: Blending `mariobros` (M) and `icarus` (I). Median, maximum generation times were 25s, 1.9m.

3.3. Multi-Game Blending Within Levels

In this application, we generate levels that contain multiple games. This is done by using custom game grids, which can cause the tiles, patterns, and reachability to change from location to location. The tag grids used are

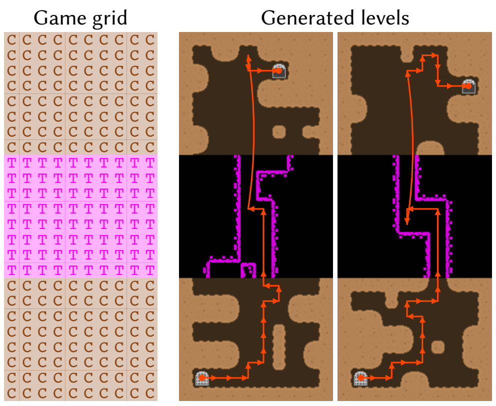

Figure 13: Blending `cave` (C) and `tomb` (T). Median, maximum generation times were 2s, 2s.

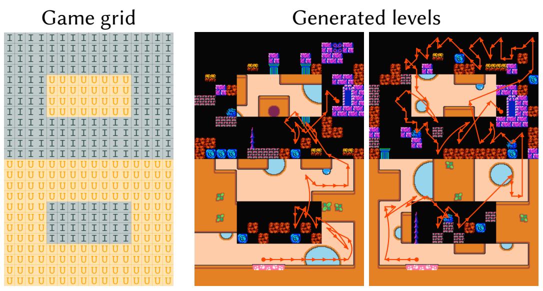

Figure 14: Blending `supercat` (U) and `icarus` (I). Median, maximum generation times were 1.1m, 5.0m.

either default tags or void tags to carve out a shape for the level. In this application, we processed all the games' example levels (except `zelda`, which was not used) at once.

Figure 12 shows a blend of `mariobros` and `icarus`, with two horizontal `mariobros` sections separated by a vertical `icarus` section (the asterisk tag allows any tile to be placed). Both games have the same `platform` reachability throughout. Figure 1 is similar, but adds a section of `cave` in between the two games. Interestingly, even after leaving the `cave`, the `platform` reachability template can then stand on the top of `cave` tiles to jump. Some partial `mariobros` flags are also placed.

Figure 13 shows a blend of `cave` and `tomb`, with two sections of `cave` separated by a section of `tomb`. Since the reachability template is determined by the game at the source of the edge, when leaving the `tomb` section, the player slides until hitting a `cave` wall. Also, since `tomb` uses black tiles for both the open play area and the non-playable surrounding area, in the transitions between games it can be unclear what area is open to the player.

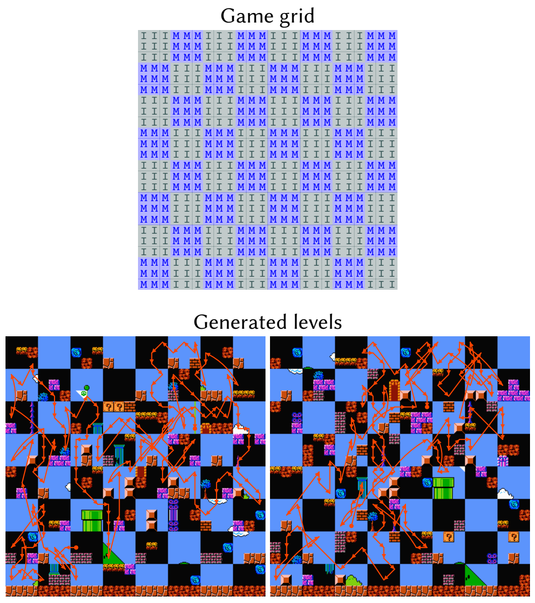

Figure 15: Blending `mariobros` (M) and `icarus` (I) in a checkerboard pattern. Median, maximum generation times were 3.8m, 19.9m.

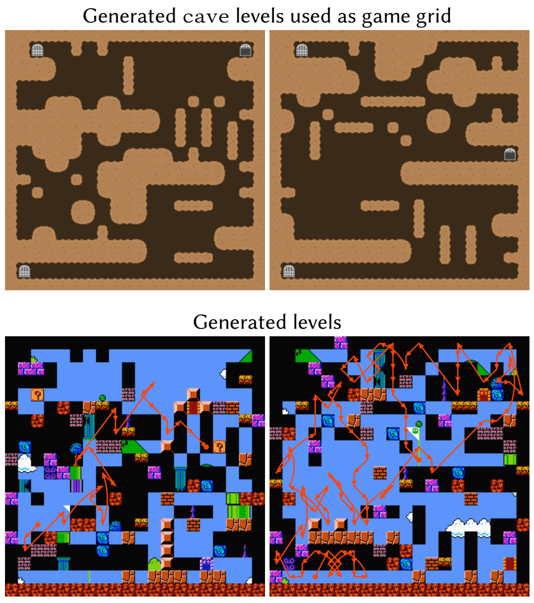

Figure 16: Generating `cave` levels, and then using those as the game grid to generate blends of `mariobros` and `icarus`. Median, maximum generation times were 2s, 2s for the `cave` levels and 10s, 1.1m for the `mariobros` / `icarus` levels.

Figure 14 shows a blend of `icarus` and `supercat`, where each game's section has a block of the other game in it, making the transition between games more frequent. Both games share walking and falling, but the types of jumps available change from game to game.

Figure 15 shows a checkboard of `mariobros` and `icarus` with frequent changes between games. Perhaps unsurprisingly, there is not much overall coherent structure to the level, and the generator does not place clear goal structures (flagpole, door) at the end of the level.

Figure 16 shows a two-step generation process that blends three games in a different way. First, a `cave` level is generated (without any reachability requirements). Then, it is converted into a game grid with the walls becoming `icarus` and the rest becoming `mariobros`. This game grid is then used to generate a level. Similar to the checkerboard levels, the frequent game changes appear to prevent the level from having a clear structure.

When blending multiple games in a level, it does appear that switching games too frequently may not be desirable, and can obscure the goal of the level.

4. Conclusion

In this work we presented an approach to constraint-based game blending, that uses a *game* grid that can specify tiles, tile patterns, and reachability to be used by the constraint solver on a per-location basis in the grid. We showed three applications: using one game's level layout with another game's tile patterns, generating levels using the tile patterns from one game and the reachability from another game, and generating levels that combine multiple games in a single level.

We showed game blends that produced output in a subjectively reasonable amount of time (a few seconds to a few minutes). Some work is necessary to find settings that produce output within a desired timeframe; however, once those settings are found they can be used to generate many levels. Some example blends we tried did not seem to work well. For example trying to generate a tall `zelda` level with `platform` reachability. It may be that in these cases it was difficult for the solver to find a way to fit through the small openings between `zelda` rooms.

We would like to be able to play the generated blended level, and the changing movement within a level. This would need to deal with unclear goals in blended levels. Potentially, a level designer could indicate certain arrangements of tiles in the examples as needing to be maintained in their entirety in the generated levels.

While we used the game grid to specify games, it could be used to specify other collections of tiles and patterns within a game, such as different example levels within the same game, or possibly different styles or even difficulties (if these could be captured in tiles and patterns).

It may also be interesting to allow multiple reachability templates simultaneously, to generate levels that can be played in multiple games at the same time.

Currently, the game grid is fully specified as an input to the generator. However, we would like to give the generator more flexibility in where games are placed, for example, leaving some locations unspecified and letting the generator fill them in. Then it may be possible to, for example request a level that starts as `mariobros` and ends as `icarus`, but let the generator decide exactly how the transition happens.

Acknowledgments

The authors would like to thank Colan Biemer and Anurag Sarkar for their feedback.

References

[1] S. Cooper, Sturgeon: tile-based procedural level generation via learned and designed constraints, Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment 18 (2022) 26–36.

[2] M. A. Boden, The Creative Mind: Myths and Mechanisms, 2nd ed., Routledge, London; New York, 2004.

[3] N. Shaker, J. Togelius, M. J. Nelson, Procedural Content Generation in Games, Springer International Publishing, 2016.

[4] M. Gumin, Wavefunctioncollapse, https://github.com/mxgmn/WaveFunctionCollapse, 2016.

[5] P. Merrell, D. Manocha, Model synthesis: A general procedural modeling algorithm, IEEE transactions on visualization and computer graphics 17 (2010) 715–728.

[6] A. Summerville, S. Snodgrass, M. Guzdial, C. Holmgård, A. K. Hoover, A. Isaksen, A. Nealen, J. Togelius, Procedural Content Generation via Machine Learning (PCGML), IEEE Transactions on Games 10 (2018) 257–270.

[7] M. J. Nelson, A. M. Smith, ASP with applications to mazes and levels, in: N. Shaker, J. Togelius, M. J. Nelson (Eds.), Procedural Content Generation in Games, Springer International Publishing, 2016, pp. 143–157.

[8] I. Karth, A. M. Smith, WaveFunctionCollapse is constraint solving in the wild, in: Proceedings of the 12th International Conference on the Foundations of Digital Games, 2017, pp. 68:1–68:10.

[9] S. Snodgrass, S. Ontañón, Controllable procedural content generation via constrained multi-dimensional Markov chain sampling, in: Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, 2016, pp. 780–786.

[10] A. Sarkar, Z. Yang, S. Cooper, Controllable level blending between games using variational autoencoders, in: Proceedings of the Experimental AI in Games Workshop, 2019, p. 8.

[11] A. Sarkar, S. Cooper, Dungeon and platformer level blending and generation using conditional VAEs, in: 2021 IEEE Conference on Games, 2021, pp. 1–8.

[12] M. Guzdial, M. Riedl, Combinatorial creativity for procedural content generation via machine learning, in: Proceedings of the First Knowledge Extraction from Games Workshop, 2017.

[13] M. Guzdial, M. Riedl, Learning to blend computer game levels, in: Proceedings of the Seventh International Conference on Computational Creativity, 2016.

[14] S. Snodgrass, A. Sarkar, Multi-domain level generation and blending with sketches via example-driven BSP and variational autoencoders, in: International Conference on the Foundations of Digital Games, 2020, pp. 1–11.

[15] A. Summerville, A. Sarkar, S. Snodgrass, J. C. Osborn, Extracting physics from blended platformer game levels, in: Proceedings of the Experimental AI in Games Workshop, 2020.

[16] A. Sarkar, S. Cooper, tile2tile: learning game filters for platformer style transfer, Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment 18 (2022) 53–60.

[17] A. Ignatiev, A. Morgado, J. Marques-Silva, PySAT: a Python toolkit for prototyping with SAT oracles, in: Theory and Applications of Satisfiability Testing – SAT 2018, 2018, pp. 428–437.

[18] A. Ignatiev, A. Morgado, J. Marques-Silva, RC2: an efficient MaxSAT solver, Journal on Satisfiability, Boolean Modeling and Computation 11 (2019) 53–64.

[19] M. H. Liffiton, J. C. Maglalang, A cardinality solver: more expressive constraints for free, in: Theory and Applications of Satisfiability Testing – SAT 2012, 2012, pp. 485–486.

[20] Kenney, Free game assets, https://www.kenney.nl/assets, 2022.

[21] Nintendo, Super Mario Bros., 1983. Game [NES].

[22] A. J. Summerville, S. Snodgrass, M. Mateas, S. Ontañón, The VGLC: The Video Game Level Corpus, arXiv:1606.07487 [cs] (2016).

[23] A. J. Summerville, S. Philip, M. Mateas, MCMCTS PCG 4 SMB: Monte Carlo tree search to guide platformer level generation, in: AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, 2015.

[24] Nintendo, Kid Icarus, 1987. Game [NES].

[25] Neutronized, Super Cat Tales, 2016. Game [iPhone].

[26] Happymagenta, Tomb of the Mask, 2016. Game [iPhone].

[27] Nintendo, The Legend of Zelda, 1986. Game [NES].