Casual Creation of Tile Maps via Authorable Constraint-Based Generators

Dan Carpenter, John Thomas Bacher, Henry Crain, Chris Martens
Department of Computer Science, North Carolina State University
{dcarpen2, jtbacher, hrcrain}@ncsu.edu, martens@csc.nscu.edu

CEUR-WS.org/Vol-3217/paper18.pdf

Abstract

Tile-based maps are used in a wide variety of games, including *Zelda: Link's Awakening*, *Super Mario Bros*, and many tabletop role-playing games. They represent a relatively simple method for designing game worlds, which makes them a great medium for casual users. However, creating tile maps entirely by hand can be tedious, especially for large maps or across several map generation tasks. There is potential for generative systems to support users in more efficiently and enjoyably creating tile maps, but a more streamlined content generation experience often comes at the cost of reduced control over the generated content. In this paper, we present a mixed-initiative map generation tool that supports users in creating customized tile maps by providing complete control over both the map being generated and the rules governing an underlying Answer Set Programming-based generator. Using a small palette of rules designed for tile map generation, users define constraints on the types of maps the system will generate, view and edit generated maps, and selectively re-generate sections of the map until they are satisfied. We discuss our current progress on this tool and present several opportunities for future work, with a particular focus on expanding the capabilities for authoring rule-based tile map generators.

Introduction

Tile-based maps are used in many video games, including *Zelda: Link's Awakening* and *Super Mario Bros*, and tabletop role-playing games like *Dungeons and Dragons* and *Pathfinder*. There is extensive support for the creation of tile maps and the use of tile maps for creating video games. For example, the Tiled map editor (Thorbjørn Lindeijer 2008) has been used in the creation of hundreds of indie games, and the popular Unity game engine (Unity Technologies 2005) features native support for tile maps. For tabletop role-playing games, map editors like Tiamat Tile Mapper (RPGObjects.com 2010) and map generators like DunGen (DungeonChannel.com 2019) are useful tools for creating tile maps. Yet, despite the substantial support for the use of tile maps in games, there is a lack of tools that leverage the full capabilities of procedural content generation to support the creation of tile maps. Often, tile map creation tools either require users to manually edit every tile on the map or allow them to configure a small set of parameters and then repeatedly generate new maps until they find something they like. There is significant potential for a tool that combines the benefits of both tile map editors and generators, allowing users to directly edit the map while also enabling collaboration between the user and a generative system to improve efficiency and foster creativity.

Mixed Initiative Co-Creation (MI-CC) systems aim to foster this type of productive human-computer collaboration during creative tasks (Yannakakis, Liapis, and Alexopoulos 2014). By supporting interaction between a human initiative and a computer initiative, with both the human and computer proactively contributing to a creative task, MI-CC systems can lead to truly co-creative experiences. In an MI-CC relationship, the human and computer are able to work in harmony to complement each other's weaknesses. For example, the computer can act as an expert guide for a particular content creation task by providing suggestions based on an existing body of work, thereby scaffolding a user's initial exploration of a new domain. The computer can also take on the role of an additional worker providing suggestions for parts of the generated content that the user is less interested in focusing their time on.

MI-CC tools can be designed to embody the principles of casual creators (Compton and Mateas 2015), which are accessible to users with varying levels of technical knowledge and domain expertise and prioritize the experience that users have while exploring a design space in collaboration with a generative system. A casual creator is a tool that allows a user to rapidly explore a range of possibilities and discover unusual or novel results. They are meant to be enjoyable to use; the user feels a sense of pride in what they have discovered. People can engage with them casually in the sense that they do not require any background knowledge or training.

In this paper, we present a mixed-initiative tile map creation tool that implements several casual creator patterns (Compton and Mateas 2015) and provides users with a simple yet powerful set of controls over the generation of a map and the underlying generative system. The tool is built around an Answer Set Programming (ASP) constraint-based generator that the user can modify through a set of natural language rules that define characteristics of the maps that will be generated. ASP was used for constraint solving because of its extensive use in procedural content gen-

Copyright © 2021for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

eration tasks, including puzzle and level generation (Smith and Mateas 2011; Neufeld, Mostaghim, and Perez-Liebana 2015). Yet, ASP is not very approachable for non-technical users, so we sought to provide some of the content generation power of ASP with a more approachable graphical user interface. In our tool, users construct rules by selecting options from dropdown menus and typing values into input fields, thus ensuring the syntactic validity of each rule. These rules are then translated into ASP and used to generate tile maps that meet the desired specifications. Rulesets can be saved and loaded, allowing users to design different map generators that they can later refine to meet specific scenarios, as well as providing a mechanism for sharing map generators between users. Additionally, users can directly modify the map that is being created by adding specific tiles or locking generated sections of the map that they want to keep. This allows users to exercise as much control over the map as they desire while supporting iterative refinement in collaboration with the generator.

We discuss our progress on the tool's design and development, describing our initial goal of supporting the creation of individual tile maps and our recent focus on supporting the creation of rule-based tile map generators. We also present promising directions for future work, especially opportunities for expanding the tool's rule authoring capabilities.

Related Work

Mixed-initiative and casual creator tools have been developed to support a wide range of content generation tasks, including the creation of game maps and levels with Tanagra (Smith, Whitehead, and Mateas 2011), Evolutionary Dungeon Designer (Alvarez et al. 2018), and Sentient Sketchbook (Liapis, Yannakakis, and Togelius 2013); entire games, including entities, relationships, and mechanics with Germinate (Kreminski et al. 2020); and characters and items for tabletop role-playing games with Imaginarium (Horswill 2020). Tanagra, Evolutionary Dungeon Designer, and Sentient Sketchbook present approachable interfaces that allow users to interact directly with the content that is being generated and receive support or inspiration from a generative system. For example, in the Evolutionary Dungeon Designer, users manually edit tile-based dungeon maps, placing floor, wall, enemy, and treasure tiles on a grid. At any time, users can view suggested versions of the dungeon that an evolutionary algorithm has generated based on the dungeon's current configuration. By taking inspiration from the suggested dungeons, users are able to explore designs that they may not have considered otherwise.

Germinate and Imaginarium differ from Tanagra, Evolutionary Dungeon Designer, and Sentient Sketchbook because they support users not only in generating artifacts, but more broadly in the creation of the PCG systems that generate those artifacts. These casual creators present novice-friendly interfaces for declaring constraints on generated artifacts, thereby allowing users to create their own procedural content generators.

Germinate, which is built on top of the Gemini game generator (Summerville et al. 2018), presents a domain-specific visual programming language that allows users to declare constraints on the types of games that the system will generate. These constraints include types of entities (i.e., graphical objects, such as characters), resources (i.e., quantitative values that define the goals of the game, such as happiness or confidence), relationships between entities and resources, and triggered gameplay events. Constraints in the visual programming language are translated by Germinate into Gemini intents, which are ASP programs. When users are done declaring constraints, they ask Germinate to produce a batch of games that attempt to satisfy the constraints specified by the user while also introducing additional entities, resources, relationships, or triggers. The generated games are presented in a playable form, so users can immediately playtest what was created based on their core design intent. Additionally, the games are displayed in the same visual programming language that is used to create the design intent for generating games. This allows users to clearly see where their constraints were implemented or modified and to take inspiration from generated games by pulling new constraints into their core design intent.

Imaginarium supports users in the development of procedural content generators for tabletop role-playing characters and items. Users provide structured natural language descriptions of some rules that they want a set of entities (e.g., characters and items) to follow and the system generates entities meeting these requirements. For example, by enumerating sets of possible characteristics that cats could have (e.g., "Cats are white, gray, black, or ginger"), followed by asking Imaginarium to "Imagine 3 cats", the system will generate three cats that embody some combination of the possible characteristics. This is done by translating the structured natural language the user provides into an ASP program, running the program, and translating the output back into natural language. By doing so, users are able to leverage some of the power of ASP without needing to be proficient in logic programming. This allows the generative tool to be approachable for users with many different backgrounds, since the technical barrier to entry has been significantly reduced.

Beyond the work presented in Germinate and Imaginarium, several other domain-specific languages have been developed to support the creation of generative systems, such as the Grammatical Item Generation Language (GIGL; Chen and Guy 2018), Ceptre (Martens 2015), and Tracery (Compton, Filstrup et al. 2014). While not necessarily designed to provide casual users with access to PCG system creation (e.g., GIGL creates generators that interface with C++), GIGL and Ceptre significantly reduce the effort required to implement generative systems. With these languages, users can more rapidly prototype new procedural content generators or games that rely on these generative systems. Tracery, on the other hand, was designed with casual users in mind and supports the creation of grammar-based story generators. By writing a marked up story similar to a Mad Libs (e.g., "The boy saw the #animal# at the zoo") and a set of replacement rules (e.g., animal $\rightarrow$ [giraffe, monkey, zebra]), users are quickly able to create systems that can generate a wide range of interesting stories.

The tool that we present in this paper was originally in-

spired most heavily by the MI-CC systems that are concerned with generating individual game artifacts (i.e., Tanagra, Evolutionary Dungeon Designer, and Sentient Sketchbook). These systems inspired the high level of interactivity that our tool provides to users when they are generating a tile map. However, we have recently identified the use of ASP-based rules to design tile map generators as a particularly interesting application of our tool. Thus, our focus has begun to shift toward developing a system that has more in common with Germinate, where the central interaction being supported is the use of a domain-specific programming language to define procedural content generators.

Current Progress

We have made significant progress toward developing a mixed-initiative casual creator tool that helps users make tile maps. The tool, developed with the Unity game engine, features a map generator based on Answer Set Programming that can be configured by a set of natural language constraints. There is also an interactive representation of the map that the user can edit at any time. By combining an interactive map with a configurable constraint-based generator, users are able to manually add whatever they want to the map and ask the generator to fill in all of the parts that they are less concerned about. Once they see what the generator creates, the user can pick and choose any parts that they like and ask the generator to once again fill in the gaps. Through this iterative process of co-creation between the user and the generator, users have the benefits of complete control over the generated map while also having a reduced workload and the opportunity to be inspired by the generator.

The tool's interface is broken up into five different areas, which are shown in Figure 1. Area 1 displays the map that is being collaboratively generated by the user and the ASP-based map generator. The map is interactive, allowing users to manually add tiles in specific locations and to lock certain areas of the map to let the generator know what it is allowed to overwrite. Area 2 allows users to start creating a new map and to assign a name to the map. Clicking the button to create a new map also brings up a window where users specify the desired height and width of the map. Area 3 shows all of the tiles that are available to the user and the generator. Currently, users only have access to a small set of tiles, but a future version of this tool will allow users to import their own images as tiles. Users can click to select tiles in this area and manually place them on the map. They can also right-click tiles in this area to assign tags, which allow users to provide metadata about tiles to group them together. For example, house, palace, and temple tiles might all be grouped together under a "building" tag. Area 4 is the rule creation interface, which provides six rules (explained in detail in the next section) that can be used to create tile map generators. These rules can operate on both tile types and tags. Finally, Area 5 has a "Generate new map" button that users can click to ask the generator to create a new map or re-generate the unlocked portions of the map that they are currently working on.

Rules for Map Generation

The tool relies on Answer Set Programming (ASP), specifically using the Clingo language (Gebser et al. 2008). Answer Set Programming allows users to specify atoms, constraints, and rules which the solver abides by and tries to generate a valid set of atoms that meet those constraints. However, while ASP is a powerful tool for constraint-based generation (Smith et al. 2012), the complexities of ASP and first-order logic make it largely inaccessible to casual users. Therefore, we sought to translate the complex syntax of ASP into simple natural language phrases that a user can use to craft tile map generators.

Rules were created in two directions. First, we formulated the kinds of constraints we would expect from a generator, which included things like specifying the types and how many of each building should exist, the proximity of buildings to other buildings, and the connections between buildings. From those formulations, we implemented an example of each in Clingo, our chosen ASP language, in order to establish that it was possible to create the rule in a generalizable way. From there, the only remaining step was to create an internal C# script that queries Clingo with a file (or, as we came to call it, a rule set) and awaits the response in the form of logic atoms which indicate the layout of the map.

We chose to formulate rules as natural language sentences where the user is able to adjust numbers and select operative words to form legal rules (see Figure 2 for some example rules in the tool). Each rule has an associated C# function which accepts the numbers and operative words as parameters and returns all necessary Clingo code for that specific rule as a string. When the user indicates they are ready to generate a new map with their existing rules, the proper function calls are made and all the Clingo code is added to a base Clingo file which contains definitions about the domain space. The result of solving the Clingo file produces the logic atoms which are parsed and displayed on the screen.

Users construct rules by typing strings or numbers into input fields or by selecting values using dropdown menus. Rules are validated before being compiled into ASP code, so individual rules that get passed to the generator are always error-free. In fact, the only way for the user to produce invalid rules is to leave some input fields blank, but this does not affect the generator because incomplete rules are ignored. However, it is not always true that the rules are valid when taken together, since there could be contradictory constraints. The tool notifies users when a map cannot be generated given the rules and locked tiles on the map, but it does not currently do anything to highlight the potential problems.

Base Rules A base rule set contains all information that Clingo needs to know about two-dimensional tile maps, such that it is able to generate valid maps. The first element of the base rule set is the set of dimension atoms. These define that there are two dimensions, X and Y, which each take a numeric input. The dimensions define the size of the map. The second element is a choice rule which indicates to Clingo

Figure 1: The tool's interface, with five distinct areas: (1) the interactive map, (2) a button for starting the creation of new maps, (3) the tiles available to the user and the map generator, (4) the rules that define the map generator, and (5) a button for generating new maps or re-generating the unlocked portions of the current map.

that there should be exactly one tile on each dimension pair. Finally, the concept of adjacency is established by creating atoms that define vectors of movement. These step rules are expanded to arbitrarily large X and Y coordinates based on the rules used during generation. The base rule set is never exposed to the user, so they do not need to worry about creating valid maps and can focus on defining rules that generate interesting maps.

Rules Presented to the User There are five main rules available to the user for crafting a tile map generator: Adjacency, Proximity, Connection, Count, and On. The natural language and Clingo representations for these rules are laid out below. A sixth rule (Tile Inclusion) allows users to specify which tiles the generator can use when creating a map, thus providing a way to prevent irrelevant tiles from appearing in certain maps (e.g., ocean tiles in a landlocked city).

The Count rule is the simplest of all the rules. It states "There are $(\geq, \leq, =)$ *Count TileA*". The rule translates to the following ASP code:

```
:- #count{X, Y : at(X, Y, house)}
  (<, >, !=) Count.
```

A Count rule is satisfied if there exists a number of the specified building or tag to satisfy the rule. The count rule can be used to indicate a specific number of buildings that should exist, or to create a range for a certain type of building.

The Adjacency rule specifies that a certain tag or tile must have a certain number of another tag or tile adjacent to it. It states "There are $(\leq, \geq, =)$ *Count TileA* adjacent to each *TileB*." The rule translates to the following ASP code:

```
:- at(X, Y, TileA),
  #count{Z, N : at(Z, N, TileB),
    steplong(DX, DY, 1),
    Z = X + DX, N = Y + DY}
  (<, >, !=) Count.
```

For example, the rule "There are at least four parks adjacent to each palace," indicates that wherever a palace is placed, there should be four parks in the surrounding eight squares.

The Proximity rule is an extrapolation of the Adjacency rule for a specified radius around the specific tile. It states "There are $(\leq, \geq, =)$ *Count TileA* within *Dist* spaces of each *TileB*". The rule translates to the following ASP code:

```
:- at(X, Y, TileA),
#count{Z, N : at(Z, N, TileB),
  steplong(DX, DY, Dist),
  Z = X + DX, N = Y + DY}
(<, >, !=) Count.
```

Using the Proximity rule instead of the Adjacency rule, our previous example could be accomplished like this: "There

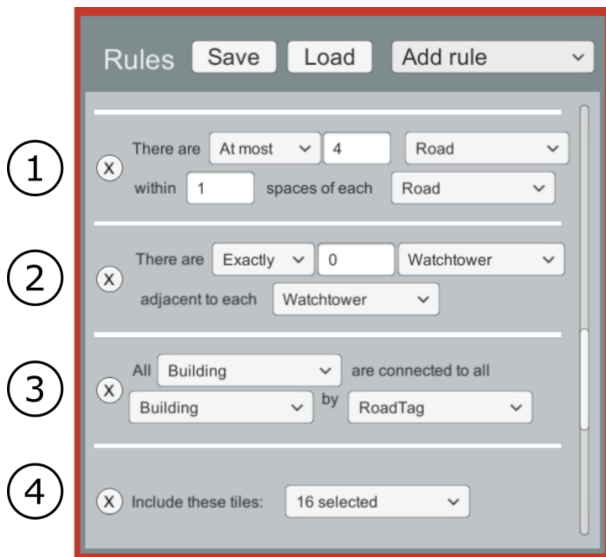

Figure 2: Examples of (1) a Proximity rule, (2) an Adjacency rule, (3) a Connection rule, and (4) a Tile Inclusion rule in the tool.

are at least four parks within one space of each palace."

The On rule directly uses the dimensions in the base rule set to specify whether a tile should or should not be on a specific coordinate. It states "All tiles in Row/Column *Num* are *TileA*". For specifying tiles in a certain row, the rule translates to the following ASP code:

```
:- not at(X, Num, TileA).
```

As an example, the maps shown in Figure 3 use an On rule to ensure that only tiles with a Wall tag appear on the outer edges of the map, thus creating a walled city.

The last rule, and the most complicated, is the Connection rule, which states "All *TileA* are connected to all *TileB* by *TileC*". The rule translates to the following ASP code:

```
has_TileC.
1 {TileC_start(X,Y) : dimX(X), dimY(Y),
  at(X, Y, TileC)} 1 :- has_TileC.
TileC_conn(X, Y) :- TileC_start(X, Y).
TileC_conn(NX, NY) :- TileC_conn(X, Y),
  step(DX, DY), NX = X + DX, NY = Y + DY,
  at(NX, NY, TileC).
:- at(X, Y, TileC), not TileC_conn(X, Y).
:- at(X, Y, TileA),
  #count{Z, N : at(Z, N, TileC),
  step(DX, DY),
    Z = X + DX, N = Y + DY} < 1.
:- at(X, Y, TileB),
  #count{Z, N : at(Z, N, TileC),
  step(DX, DY),
    Z = X + DX, N = Y + DY} < 1.
```

The Connection rule takes three tiles or tile tags as parameters and specifies that every instance of the first tile is connected to every instance of the second tile by the third tile. Take, for example, the road Connection rule which states that "All Buildings are connected to all Buildings by Roads." This guarantees that there is a connected chain of roads that links buildings to one another. It implies, as well, that, "All roads are connected to all roads by roads." The third tile in the rule will always be connected to itself in order to achieve the rule.

Intended Workflow

The workflow we intend to support with this tool consists of two tasks: using rules to define a tile map generator, then iteratively collaborating with the generator to create specific map instances.

The creation of a tile map generator would proceed as follows:

1. Define constraints on the map generator by adding new rules or by loading an existing rule set.
2. Ask the generator to create one or more new maps.
3. Inspect the generated maps and refine the set of rules that make up the map generator to better reflect your design goals.
4. Repeat from Step 2 until the generator consistently creates maps that you are satisfied with.

Once the generator is more or less finalized, the user would take the following steps to create a specific tile map instance:

1. Specify the dimensions of a new map.
2. Ask the generator to create new maps until you find one that interests you.
3. Manually edit the map by adding tiles or locking sections of the generated map that you want to keep.
4. Ask the generator to re-generate the unlocked parts of the map (see Figure 3 for an example).
5. Repeat from Step 3 until you have a map that you like.

Evaluation as a Casual Creator

Since the primary purpose of this tool is to support a wide range of users in the creation of tile maps and tile map generators, it is important to evaluate it in relation to Compton and Mateas' (2015) eleven design patterns for casual creators.

Instant feedback. The tool features an interactive display of the map being created, which is instantly generated when the user asks for a new map. In our limited testing the generator has always returned a result instantly, but it is conceivable that some rule sets could cause the generator to slow down. To help mitigate any issues related to slow generation, we have configured the tool so an asynchronous call to the Clingo generator is started whenever a rule is added or changed. Thus, when users ask the tool to generate a new map, the tool has already been working behind the scenes to generate that map.

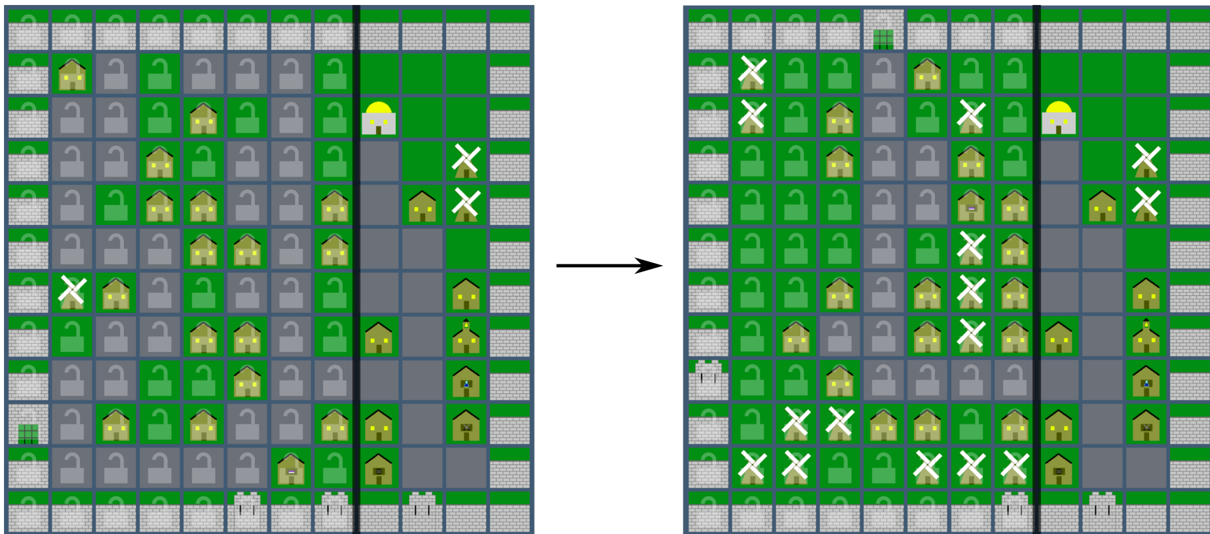

Figure 3: An example of two generated maps with the same locked section. The section to the right of the black line is locked; the generator has filled in the rest with possible arrangements of tiles that satisfy the constraints given (not shown).

Chorus line. With a larger focus on supporting the creation of tile map generators, and not just individual tile maps, we will want to support the chorus line pattern by allowing users to simultaneously generate multiple maps with the same generator. Thus, users will be able to get a feel for the breadth of the design space their generator covers.

Simulation and approximating feedback. Since the tile maps generated by the tool are not designed for any specific use case, such as a particular game, it is difficult to determine what metrics might be generally useful to include. Therefore, this design pattern is not addressed.

Entertaining evaluations. This is not applicable to the current version of the tool because the tile maps being generated do not undergo any sort of evaluation.

No blank canvas. As a starting point for users, we designed three base rule sets that create a city, a walled city, and a port city. These rule sets can be used to generate reasonably interesting city maps with the click of a button, and both the map and the rule sets can be modified by the user to start creating something new. Moreover, with the ability to save and share rule sets, there is potential for users to have community-sourced resources that prevent them from having to deal with a blank canvas.

Limiting actions to encourage exploration. When modifying an individual map, users have access to just two core actions - they can place tiles on the map and lock or unlock tiles to tell the generator which parts of the map it is allowed to overwrite. To define rules for a tile map generator, users only have access to five different rules, which represents a significant simplification compared to creating a generator directly using ASP.

Mutant shopping. By allowing users to directly interact with the tile map that is being generated and to lock or unlock parts of the map to tell the generator what it is allowed to overwrite, users are able to iteratively keep newly generated sections of the map that they find interesting. There is currently no equivalent feature for designing tile map generators, but in the future it may be possible to suggest new rules, either randomly or in a data-driven manner.

Modifying the meaningful. The base rules that are used to define a generic tile map generator, such as a rule for populating each grid on the map with exactly one tile, are entirely hidden from the user. The user only needs to worry about high-level rules for generating different types of tile maps.

Saving and sharing. All constraints for a map generator can be saved to a JSON file, which can be loaded back into the tool at a later time or shared with other creators. These constraints include all rules as well as the types of tiles allowed in the map and tags used to group tiles. In the future, we also plan to allow users to export individual maps that they have created as JSON files and images.

Hosted communities. We do not currently have hosted communities, but the ability to export map generator rule sets as JSON files would support this. There is a lot of potential for users to share generators that reflect different map archetypes, such as the walled city that we provide as a base rule set, that users can download and immediately use or modify to fit their needs.

Modding, hacking, teaching. We currently do not provide support for modding or hacking the tool, but it may be interesting to provide advanced users with an interface for defining their own rules templates that compile into ASP.

Future Work

Moving forward with this tool, we are most interested in expanding the rule creation interface to provide users with

greater capabilities for creating tile map generators. However, as this is meant to be a casual creator tool, it will be important for the tool to remain novice-friendly while also allowing advanced users to create more complex and expressive generators. To this end, we are interested in reimplementing the rule creation interface using Blockly, which has been used extensively in introductory programming environments (Fraser 2015). Specifically, it may be possible to leverage existing work that has made preliminary progress toward implementing Blockly in Unity-based applications (Taylor et al. 2019). Among many potential benefits, this will provide users with access to variables and arithmetic operators, which introduces significant potential for much more expressive tile map generators. For example, if a user wants to write a generalizable rule that places walls around the outside of a city, they need to know what the width and height of the map are. Additionally, having the standard look and feel of a popular visual programming language could allow users with some Blockly experience to immediately feel comfortable with our tool.

It will also be helpful to introduce debugging support in the form of static program analysis. For example, if a user adds two rules that contradict each other, the system could flag these rules and notify the user that they are not compatible. A meta analysis of the ASP program constructed by the user, such as by using Gebser et al.'s `spock` system (Gebser et al. 2007), is a promising direction for helping users understand why the set of rules they have created may be unable to generate valid maps. Compared to our current approach, which only uses dynamic program analysis to produce generic error messages based on the compiled ASP program's satisfiability (e.g., "A map could not be generated using this ruleset. Please check for errors."), providing more immediate and localized feedback would likely be beneficial to users.

Another interesting direction for future work is to investigate methods for inferring constraints from hand-authored maps (De Raedt, Passerini, and Teso 2018) rather than requiring users to manually generate every rule. Users who are more comfortable interacting with the map than authoring rules would be able to provide examples of the types of maps they wish to generate and could select rules inferred by the system that reflect their design intent. Additionally, we might want to allow users to ask the system to generate some rules for them. These rules could be randomly generated to encourage exploration in potentially novel design spaces, or they could embody some machine learned intent based on previous users' interactions with the system. Thus, we could make progress toward our original goal of having users specify only what they are interested in and allowing the computational system to fill in the gaps, only now this would be done in the space of map generator rules, not tiles on an individual map. Alternatively, it would be interesting to investigate the use of WaveFunctionCollapse (WFC) to allow the system to learn from hand drawn examples of maps that users want the system to emulate. WFC has demonstrated an impressive ability to extrapolate from examples of tiled images (Karth and Smith 2021), which makes it very relevant to this work. However, WFC only infers local constraints that are similar to our system's Adjacency and Proximity rules. As a result, maps created by a WFC-based generator would not be as expressive as our system allows (namely, global constraints like Count and Connection would not be natively supported). Perhaps a combination of WFC-based map generation and machine learning-based constraint inference could be implemented that takes advantage of both of these promising directions for future work.

There are also several potential opportunities for expanding the tool's generative capabilities. First, we could allow local rules that only apply to certain parts of the map, thus allowing for finer control over the maps being generated. We could achieve this by supporting the composition of maps from multiple smaller maps, each with their own generators. Using a similar approach, it would be interesting to allow users to create rulesets for different layers of a map (similar to layers in a program like Photoshop or Tiled). For example, users could generate entire game levels by first creating a map layer and then adding an entity layer on top of it that generates gameplay elements like characters or items. Finally, we could add the ability to generate hex tile maps or even abstract graph visualizations, not just square tile maps. As a result, the tool could potentially be adapted for an entirely different domain like narrative generation by introducing rules about temporal sequencing (e.g., *X* happens before/after *Y*), types of actions and states (e.g., Actors can do *Action*, Actors can be *State*), and action preconditions and effects (e.g., Doing *Action* makes an actor *State*). Providing support for other domains would likely require substantial work, but the potential to expand beyond tile maps is certainly interesting.

Regardless of which directions we choose to pursue, it will be essential to conduct user studies with an updated version of the tool. We will want to evaluate how easy it is for novices to use the tool, the types of generators and artifacts that users are able to make with the tool, and the extent to which users find the tool engaging and rewarding.

Conclusion

We describe preliminary progress on a mixed-initiative tool for the casual creation of tile maps. The tool features a small palette of natural language rules specifically designed for tile map generation tasks, thereby allowing users to design their own tile map generators without requiring knowledge of the underlying ASP code that the rules are compiled into. Users are able to offload as much of the map generation task to the system as they desire, presenting opportunities for increased efficiency and inspiration, while support for direct editing of the map provides users with the ability to override the generator at any time. We evaluate the tool in relation to Compton and Mateas' (2015) design patterns for casual creators and find that it aligns with many of the patterns, but it will be important to conduct studies to better evaluate the tool's usability and usefulness. Moving forward, there are several promising opportunities for future work, including greater rule authoring capabilities using a Blockly implementation of the rule interface, improved debugging support, and expansion into content generation tasks beyond tile maps.

References

Alvarez, A.; Dahlskog, S.; Font, J.; Holmberg, J.; Nolasco, C.; and Österman, A. 2018. Fostering creativity in the mixed-initiative evolutionary dungeon designer. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, 1–8.

Chen, T.; and Guy, S. J. 2018. GIGL: A domain specific language for procedural content generation with grammatical representations. In *Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Compton, K.; Filstrup, B.; et al. 2014. Tracery: Approachable story grammar authoring for casual users. In *Seventh Intelligent Narrative Technologies Workshop*.

Compton, K.; and Mateas, M. 2015. Casual Creators. In *ICCC*, 228–235.

De Raedt, L.; Passerini, A.; and Teso, S. 2018. Learning constraints from examples. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

DungeonChannel.com. 2019. DunGen Dungeon Generator. URL https://dungen.app/dungen/.

Fraser, N. 2015. Ten things we've learned from Blockly. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, 49–50. IEEE.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Ostrowski, M.; Schaub, T.; and Thiele, S. 2008. A user's guide to gringo, clasp, clingo, and iclingo .

Gebser, M.; Pührer, J.; Schaub, T.; Tompits, H.; and Woltran, S. 2007. spock: A debugging support tool for logic programs under the answer-set semantics. In *Applications of Declarative Programming and Knowledge Management*, 247–252. Springer.

Horswill, I. 2020. A Declarative PCG Tool for Casual Users. In *Sixteenth Artificial Intelligence and Interactive Digital Entertainment Conference*, volume 16, 81–87.

Karth, I.; and Smith, A. M. 2021. WaveFunctionCollapse: Content Generation via Constraint Solving and Machine Learning. *IEEE Transactions on Games* .

Kreminski, M.; Dickinson, M.; Osborn, J.; Summerville, A.; Mateas, M.; and Wardrip-Fruin, N. 2020. Germinate: A Mixed-Initiative Casual Creator for Rhetorical Games. In *Sixteenth Artificial Intelligence and Interactive Digital Entertainment Conference*, volume 16, 102–108.

Liapis, A.; Yannakakis, G. N.; and Togelius, J. 2013. Sentient sketchbook: computer-assisted game level authoring .

Martens, C. 2015. Ceptre: A language for modeling generative interactive systems. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*.

Neufeld, X.; Mostaghim, S.; and Perez-Liebana, D. 2015. Procedural level generation with answer set programming for general video game playing. In *2015 7th Computer Science and Electronic Engineering Conference (CEEC)*, 207–212. IEEE.

RPGObjects.com. 2010. Tiamat the Tile Mapper. URL http://www.rpgobjects.com/tiamat/.

Smith, A. M.; Andersen, E.; Mateas, M.; and Popović, Z. 2012. A case study of expressively constrainable level design automation tools for a puzzle game. In *Proceedings of the International Conference on the Foundations of Digital Games*, 156–163.

Smith, A. M.; and Mateas, M. 2011. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games* 3(3): 187–200.

Smith, G.; Whitehead, J.; and Mateas, M. 2011. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *IEEE Transactions on Computational Intelligence and AI in Games* 3(3): 201–215.

Summerville, A.; Martens, C.; Samuel, B.; Osborn, J.; Wardrip-Fruin, N.; and Mateas, M. 2018. Gemini: Bidirectional generation and analysis of games via asp. In *Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference*, volume 14.

Taylor, S.; Min, W.; Mott, B.; Emerson, A.; Smith, A.; Wiebe, E.; and Lester, J. 2019. Position: IntelliBlox: A toolkit for integrating block-based programming into game-based learning environments. In *2019 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, 55–58. IEEE.

Thorbjørn Lindeijer. 2008. Tiled Map Editor. URL https://www.mapeditor.org/.

Unity Technologies. 2005. Unity Game Engine. URL https://www.unity.com.

Yannakakis, G. N.; Liapis, A.; and Alexopoulos, C. 2014. Mixed-initiative co-creativity .