Designing a Combined World and Story Procedural Content Generation Engine

Brenden Lech[1], Sasha Azad[1], Jennifer Welnitz[1], Joel Jonasson[2], Chris Martens[1]
[1] North Carolina State University, USA
[2] Blast Bit Enterprises AB, Sweden
{bglech, sasha.azad, mawellni}@ncsu.edu, joel.jonasson@blastbit.net, crmarten@ncsu.edu

Abstract

Procedural content generation (PCG) has seen relative widespread adoption in games and game development. While PCG methods are gaining traction in commercial games, there are comparatively fewer cases where PCG has been used to generate the game world, characters, and the narrative for a single game. We present our research, collaborating with a small independent game studio, on the design of a game engine designed to generate a world, characters, supporting quests and overarching narrative content for their upcoming game. We discuss our choice of game artificial intelligence algorithms, decisions made in designing our system, and our progress and implementation under the constraints and requirements given to us. We posit our system will generate a rich game world with an explainable world history for players to explore and quests to complete. We believe that our engine will meet their requirements increasing the replayability of the game, reducing the authorship burden, while providing the authors with significant control over quest and narrative structure.

Introduction

As commercial video games grow more complex and business models such as free-to-play (F2P) become more common, there is a need for tools and methodologies that can help generate the growing amount of content the games require (Hendrikx et al. 2013). A class of AI algorithms termed as Procedural content generation (PCG) algorithms is one solution to this problem. PCG algorithms can generate game content such as levels, terrain, and other objects during game development or run-time. PCG has aided a wide variety of use-cases, from generating individual quests for the player, to generating non-player characters (NPCs) with motives and emotions, or generating the terrain of entire game worlds (Hendrikx et al. 2013; Smith et al. 2011). They can reduce the authoring burden on game designers, increase the replayability (Smith et al. 2011) of games by continually generating new content during runtime, or even produce unexpected content that human designers may not have conceived of (Togelius et al. 2011).

Within the games industry, these benefits of PCG have been realized especially for world generation, with examples including games such as *Minecraft*, the *Diablo series*, and *Dwarf Fortress*, all of which employ PCG to create their terrain and dungeons (Persson and Bergensten 2011; Adams and Adams 2006; Brevik 1997). However, despite the industry's adoption of PCG, there remains a lack of commercial adoption of AI PCG strategies developed in research contexts (Van Der Linden, Lopes, and Bidarra 2013).

We collaborated with Blast Bit Enterprises, a small, independent game development studio, to design a mobile game. Their game needed to have high replayability. They required a "game and story engine," a system capable of procedurally generating both the world, quests and narratives that guide the player's gameplay experience. Such a system could also afford players more agency than a manually-authored story by allowing the player to shape the narrative through their actions. As this story engine would define the player's gameplay experience, its design constitutes a crucial part of Blast Bit's game.

We describe our approach toward solving this problem by proposing a system design for Blast Bit's game and narrative engine and providing an implementation of part of the proposed system. We discuss our software design process

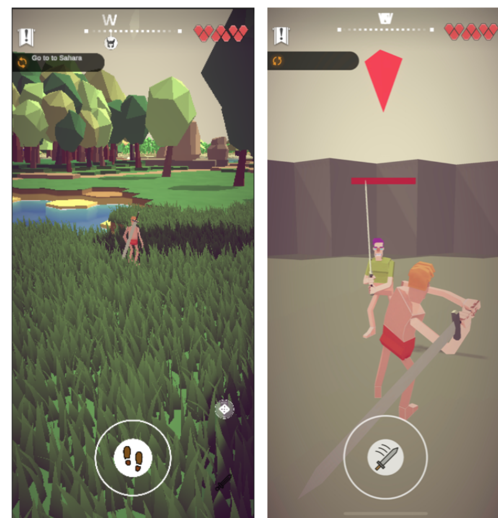

Figure 1: Gameplay depicting a player exploring the forest terrain (on the left) and battling another NPC (on the right)

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

CEUR-WS.org/Vol-3217/paper12.pdf

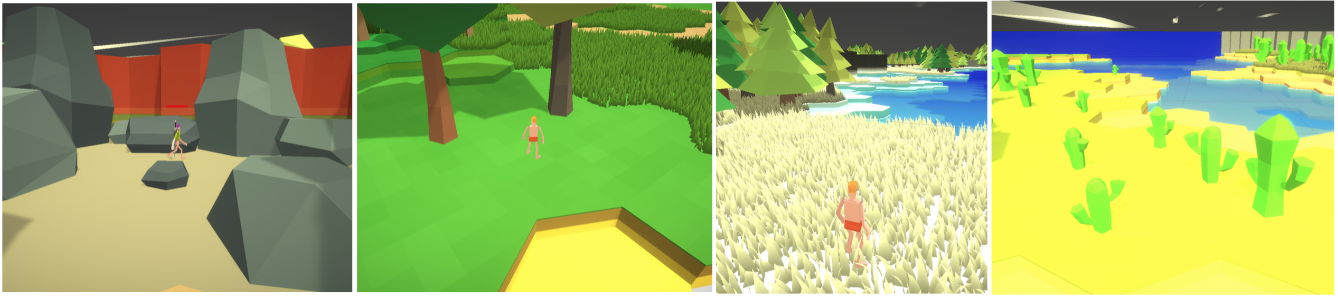

Figure 2: An example of the biomes generated in Blast Bit's Lance A Lot game. From the left this includes the quarry, forest, snow, and desert biomes.

and selection of procedural content generation algorithms and procedures working around the constraints given to us by the studio resulting in the development of their PCG Engine.

Related Work

In this section, we overview the prior use of context-free grammars (CFGs), logic programming and story systems in research and describe how they connect to our system design.

Grammar-Based Systems

Context-Free Grammars (CFGs) (Compton, Filstrup, and others 2014) have been used in prior work to generate new content (Gellel and Sweetser 2020; Dormans 2011) satisfying encoded grammar rules. While CFGs afford authors a high degree of control over the algorithm's output, they have been previously criticized as being unable to generate sophisticated stories (Black and Wilensky 1979).

Doran and Parberry (2011) detail a grammar-based approach to generating RPG quests in games. They identified common narrative and gameplay structures featured in human-authored quests and encoded these structures as rules in a CFG. While their system shows promise for generating individual quests, their generated quests take place in a relative vacuum: they have no connection to each other or other narrative structures. Additional work would be required to generate quests that take place in the context of the larger, overarching narrative structures as was required by our system.

In the domain of level generation, Dormans and Bakkes (2011) present a system that utilizes graph grammars to generate missions and level maps. However, while their system produces an interesting mission structure, producing the desired narrative structure is outside of its scope. Therefore, modifications would be needed to utilize a similar approach for generating quests with a stronger focus on storytelling.

Logic Programming

Logic programming is another promising approach, defining the world or its objects as predicates and implications (Bosser, Cavazza, and Champagnat 2010; Martens et al. 2014). Logic programming has been discussed as viable for specifying stories and analyzing their causal structure (Martens et al. 2014), generate mazes, levels of dungeon-crawler games, and in narrative planning systems (Nelson and Smith 2016; Dabral and Martens 2020) with tools such as CatSAT (Horswill 2018) allowing for the adoption of declarative programming in runtime PCG applications. While these uses have been explored separately, further work is needed to determine strategies for weaving together world and story models in ASP-based systems.

Story Systems

Narrative systems such as Versu (Evans and Short 2013) have been previously characterized as one of *strong autonomy*, or one where characters choose their actions as individual agents (Riedl and Bulitko 2013). In contrast, *strong story* systems manage NPC behavior via a centralized system such as an experience manager or drama manager (Riedl and Bulitko 2013) that dictates character behaviour to weave a cohesive story. Our work falls more on the side of "strong story" systems with our story and history generators act as a Drama Manager, dictating the actions of NPCs to advance the narrative.

Caves of Qud employs a novel method of generating unique worlds and quests, in which the world and story both are generated in stages: the first, a generic story, at the world construction, and the second when the player first enters a specific region, to finalize the details of the environment and associated story or quest (Grinblat and Bucklew 2020). Further, *Caves of Qud* includes a sophisticated history generation system, generating contextual histories, filling in the necessary details using a grammar like structure (Grinblat and Bucklew 2017). This multi-stage environment, story generation, and history generation in *Caves of Qud* are relevant to the work presented in this paper.

Finally, prior work has been done on using PetriNets, a graph based structure for narrative mediation, player choice enumeration and planning (Riedl et al. 2011; Azad et al. 2017). Our system uses a similar concept, detecting when a region-node can be activated based on meeting the preconditions of generating the biomes, quests or history for the same.

Overview of the Game and Engine

This section gives the reader an overview of the game being developed by our industry collaborators, Blast Bit. We also describe the existing game engine and the goal and requirements of the project.

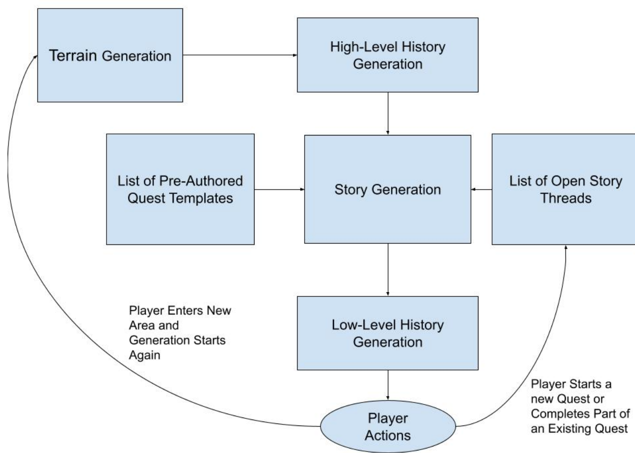

Figure 3: The system design for the story engine. Each generator feeds information into the next to generate aspects of the new game area.

The Lance A Lot Game

The game (working title "Lance a Lot") is designed for the mobile market. The game is a third-person role-playing game where the player is taken on a lighthearted journey through a fantasy world that does not take itself all too seriously. The game is created for players who enjoy exploration, interaction with hilarious characters, finding magical items and a good sword-fight. Through the journey, the player will interact with human and non-human NPCs, complete quests, fight enemies, and maybe even make choices that shape how the story unfolds.

Game Engine

The game world is divided into several separate, procedurally generated areas. Each area has its environment type, physical features, NPC characters, and associated quests. Upon entering a new area, the story engine defines the contents of that area – including characters, interactive objects, and quests – then sends this information to the world generator, which creates the physical layout of the world and populates it with the requested features.

The current game engine uses placeholder example terrain and quests. The generated information about the world state is stored in a knowledge base. Blast Bit's ontological knowledge base includes the following categories and features described below as facts about the world. We also include a few examples from each category

- Biomes: Snow, Water bodies, Quarries, Mountains, Desert (depicted in Fig. 2)
- Locative Points of Interest: Settlements, Buildings, Vegetation, Boats, Chests
- Resources: Ore, Berries, Water
- Virtual Characters:
  - By Profession - Smithy, Cooks, Kings, Knights
  - By Species - Humans, Birds, Monsters, Dragons, Wolves
  - By Relationships - Parent, Child, Spousal
  - By Physical Appearance - Arm length, head size, torso width, etc.
- Events: Earthquakes, Invasions, Siege, Theft

The knowledge base informs the game's procedural generation by putting constraints on the generation. For example, if the knowledge base states that a dragon lives near $Area0$, then the world generation knows it must create a dragon's den in an area adjacent to $Area0$. This allows for thematic consistency across areas and for information about the game's plot to be maintained, enabling larger, multi-area storylines.

Method

The designed story engine is responsible for defining the physical features and elements contained in each area, and so also constitutes a large part of world generation. Additional responsibilities for the story engine include generating historical information about the game world and populating the game world with characters. Therefore, the story engine has been divided into multiple distinct generators.

To assist in explaining the functionality of our system, we use the running example of generating a quest involving the player fighting a dragon that has been terrorizing nearby villages. We walk through the creation of this region and the selection of the quest across generators from the ground up.

We use an iterative world-story generation system. First, a terrain is generated (e.g. a mountain), with some high level history (e.g. a mountain village). Next, a story is selected to fit the world (e.g. the selection of a dragon quest). This selected story then adds further constraints on future world or terrain generation (e.g., a dragon's cave must be generated by the terrain generator, or homes in the region must be burnt down). This iterative world-story generation was chosen due in part to the vision of creating emergent stories from a simulation of the game world and its characters.

The separate generators by task, namely, the area and terrain generation, world history generation, and narrative and quest generation, have been illustrated in Fig. 3. Character generation is done by the quest and history generators. The generators work together to create each new game area, feeding information from one part of the system to the next to generate different aspects of the area. The flow of knowledge and information can be seen in Fig. 3. The triggers for the generators have been depicted as edge labels on the diagram. This section presents the current implementation of the terrain generator and describes in further detail the system design and intent of each of the generators introduced above.

Terrain Generation

When the player enters a new area, our system runs to define the area; this begins with terrain generation. The terrain generator outputs predicate-form definitions of the area's biome, natural landscape features, preliminary information about connecting areas, and any other high-level terrain definition information required.

```
isArea(Area0);
hasBiome(Area0, forest);
hasFeature(Area0, trees);
hasFeature(Area0, lake);
hasFeature(Area0, boulders);
isArea(Area1);
hasBiome(Area1, mountain);
isUndefined(Area1);
isConnected(Area0, Area1);
isArea(Area2);
hasBiome(Area2, quarry);
isUndefined(Area2);
isConnected(Area0, Area2);
```

Code 1: Example output of the terrain generator grammar. The forest biome, region $Area0$, was generated adjacent to the mountain region, $Area1$, and a quarry biome, $Area2$, which our grammar deems as valid neighbouring biomes.

The Terrain Generator utilizes Tracery (Compton, Filstrup, and others 2014), a CFG authoring tool that is intended primarily for use in text generation. Tracery afforded us with a grammar (albeit used in a non-traditional way) to define and assert the predicates and rules required to describe the game's terrain features that the BlastBit system requires.

Our generator first defines a biome that constitutes a general flora and fauna type – for instance, a forest or a mountainous region. Biomes are used to ensure that diverse environments can be generated and that consistency between adjacent areas is maintained. When defining a region, the generator either begins with a biome at random (if it is defining the first region in the world) or a predefined biome type (if it is defining a region adjacent to or following one that is already defined). Our grammar controls the adjacency rules between biomes ensuring that biomes generated adjacent to the region are realistic. For instance, a desert may not generate next to an ocean. Code 1 shows an example of the terrain generator's output.

Further, we use the Generate-and-Test methodology (Togelius et al. 2011) to generate a variety of possible neighbors using our CFG system, and select one based on constraints from the story-engine (for instance, selecting a neighboring biome that allows for a dragon's cave either in this region, or one at a specified neighboring distance from this one). For instance, while it may be possible for the CFG to generate an ocean next to a forest, our system could select either the output containing a mountain biome (to allow for the dragon's cave in this biome), or another forest biome (eventually allowing for a mountainous biome within a specified distance of, say, 3 biomes from the burnt village).

```
{"forest": [
    "hasBiome(AreaX, forest)",
    "#forest_features#",
    "#forest_connections#"],
"forest_features": [
    "#trees#",
    "#brush_chance#",
    "#boulders_chance#",
    "#lake_chance#",
    "#valley_chance#"],
"brush_chance": ["#brush#", ""],
"trees": ["hasFeature(AreaX, trees);"],
"brush": ["hasFeature(AreaX, brush);"],
"forest_connection": [
    "isArea(AreaY);"
    "hasBiome(AreaY, #f_adj_biome#);",
    "isUndefined(AreaY);",
    "isConnected(AreaX, AreaY);" ],
"f_adj_biome":
    [ "forest", "mountain",
      "grasslands", "quarry" ]
...}
```

Code 2: Code snippet from terrain-generation grammar resulting in the output shown in Code 1

In the output from our system in Code Snippet 1, the generator randomly assigned a forest biome to $Area0$. From there, the grammar expands a list of features available in forest biomes to choose which features can be found in $Area0$. In this case: trees, a lake, and boulders. This functionality will be expanded in the future to define resources available in the area, such as water, timber, and berries. Next, the terrain generator defines the biomes for adjacent regions. In Code 1, a mountain and a quarry were defined as adjacent regions to the forest. These adjacent regions are marked as "undefined" to let the system know that they must be generated further once the player leaves the current region and moves to the next. This allows us to further constrain our generation based on what the player discovers in the forest. Finally, the predicates output from the terrain generator is passed to the high-level history generator. Additionally, the predicates are taken as input by the game engine, that generates the geometric layout of the area and the locations of features within it.

High-Level History Generation

The high-level history generator is similar to the terrain generator but defines human-made structures rather than landscape features. Examples of definitions created in this generative step include those for settlements, buildings, societal histories such as war and trade, individual characters, and specific occupational and personal information about the generated characters.

While a CFG-based approach may have met many of the High-Level History Generator's requirements, many of the predicates it would generate are dependent on multiple existing predicates. For example, for a fisherman to generate, there must exist a town, and that town must trade in fish. Authoring a CFG that encodes these requirements may quickly become overly-complex and unwieldy. Other design considerations included simulating historical events and resource movement between settlements, though that level of detail was considered unnecessary for our purposes. Therefore, we decided on a design that reasons over existing predicates to generate the new predicates that describing these human-made features in the game world.

In our running example, the current area has been defined as a forest containing trees, a lake, and boulders. Because this area contains a source of water and is a temperate region, the high-level history generator reasons that a settlement here is likely and creates the new predicates $isSettlement(Town0)$ and $contains(Area0, Town0)$. From here, the generator continues to define additional, more specific information about the settlement such as the resources it trades in, the individual buildings that exist in it, and who the occupants of those buildings are. In our example, one of the buildings created is occupied by a fisherman, as the town trades in fish and is near a lake. This generation will continue until the area is sufficiently defined or until no more inferences can be made. A partial example output of this generative step can be seen in Code 3.

```
isSettlement(Town0);
contains(Area0, Town0);
tradesIn(Town0, timber);
tradesIn(Town0, fish);
isBuilding(House0);
contains(Town0, House0);
species(Person0, human);
profession(Person0, fisherman);
livesIn(Person0, House0);
...
```

Code 3: Example output of the high-level history generator

Once a history of the area and definitions of its inhabitants have been created, a preliminary model of the area's contents exists and the Narrative Generator can select a story for the area.

Narrative Generator

The narrative generator uses the predicates generated by the previously-discussed generators (shown in Code 1 and Code 3) to generate a quest for the player. The generator can ensure that the selected quest fits with the overarching story arc the player is currently participating in.

The Narrative Generator generates a story for the new area by manipulating "plot threads". When a plot thread is requested for a region, the generator either chooses from a list of existing, open plot threads from previous regions or creates a new plot thread that suits the given region. Each new

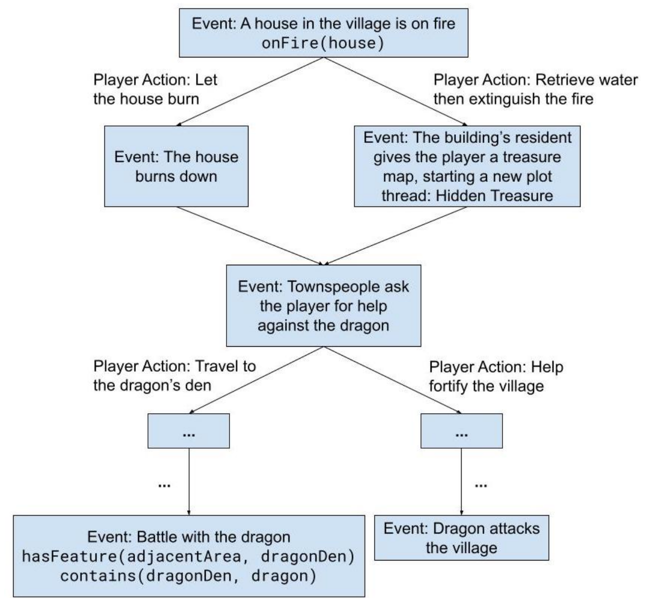

Figure 4: An authored quest template used to select new or open plot thread. Events are partially ordered.

plot thread is instantiated from a list of designer-authored quest templates.

While authoring the quest template, the designer must include preconditions (in the form of predicates) that must be met in the current world state for events in the world to occur. We define our quest template to be a tree-structure representation of a quest. Nodes in the tree are events that can occur in the region given the preconditions for the same have been met. This activation of template nodes is similar to narrative research done by Riedl et al. (2011) using PetriNets. The algorithm will detect when a region can be activated based on meeting the preconditions for the region or narrative allowing us to balance authorability against expressive generation of content.

The predicates of the quest template are generic (for instance, it defines that a house must exist to be on fire but doesn't specify which house). An example of such a quest template has been depicted in Fig. 4. Once an event occurs, the player may choose to influence the outcomes and future events of the story; expected interactions are represented by forks in the quest template's tree.

Events in the quest templates are partially ordered, for instance, the player will not encounter the dragon without first being warned of it in a prior region. This has been depicted in Fig. 4 where the player first encounters a burning home, then speaks to townspeople and hears of a dragon. Eventually, the player may traverse to an adjacent region with a dragon's den, where the dragon can be fought. These partially-ordered constraints are also returned to the terrain and high-level history generator. For instance, the history generator now knows that one of the regions connecting to $Area0$ in the future could generate townspeople that have been tormented by the dragon and communicate the same

to the player. Additionally, the terrain generator knows that in the future, once other preconditions have been met, a dragon's den must be generated high in the mountains. This further restricts the generation in future unexplored regions.

```
hasFeature(Area, lake)
adjacent(Area, adjacentArea)
hasBiome(adjacentArea, mountain)
contains(Area, town)
isSettlement(town)
contains(town, house)
isBuilding(house)
```

Code 4: An example of the prerequisites for the first event in the quest template depicted in Fig. 4

In our example, $Area0$ is the first area created, and there are no existing plot threads that can be applied. Therefore, a new plot thread must be created. To select the quest template that will be used to create the new plot thread, the Narrative Generator searches through a list of available authored quest templates whose preconditions are met by the current world definition. From this list, the Narrative Generator randomly selects a quest template, favouring those which have not recently been used to create a previous plot thread. In this case, it begins the hero's journey by selecting the dragon quest. It should be noted, that the narrative generator may choose to start another quest template even if an earlier one is incomplete. This may be seen as necessary, for instance, if the predicates of the world do not currently support an existing quest. Thus, the player may be partaking in more than one quest at a time.

From the perspective of the human designer, a single event, in this case, that of the house burning down, may be used in several quest templates. In one quest, the player may find out that the bandits in the area are responsible, in another, it could be the dragon. In the future, we will attempt to design an authoring tool that will allow for events to be tagged with a variety of quests, and accordingly, procedurally generate the characters or locations involved.

Low-Level History Generation

Low-level history generation is the final step in the story engine's generation of the area. The purpose of low-level history generation is to modify the area to better fit the plot thread selected by the story generator. This generator works backwards to create explanations for why those objects exist in the world and how they affect it. Since the selected plot has defined events that will happen and objects that must exist, the generator will add those objects to the world generated so far. This chain of reasoning continues until the area definition makes sense in the context of the objects that were added during story selection.

The low-level history generator also serves the purpose of generating history on-demand to flesh out an existing world. This is useful for satisfying Blast Bit's goal of allowing the player to ask NPCs questions about the world's history.

To continue with the dragon quest example, since a dragon exists, the generator will reason that a nearby village fears it. If a village fears something, it may build fortifications. These fortifications are added to the region. Additionally, NPCs living within a range of the dragon on the map will start to mention their fear of the dragon to the player, giving the player breadcrumbs as to what to expect. Once this history has been generated, the player can then ask NPCs of the region questions regarding why a house was destroyed, or how fortifications were broken down, improving the explainability of the generators.

For this generation, we intend to use an in-house lock-and-key logic programming language developed by Blast Bit. The predicates of the world are described as symbols that can be defined as a combination of keys. The preconditions in the story states are represented as locks and fulfilling these preconditions unlock new keys affecting the availability of our graph and change in story state.

We are excited about the capability of this generator to add to the *explainability* (Zhu et al. 2018) of the world. The output from this generator will describe a loosely built history for our world that can be used to explain the reasons for the procedurally generated content.

Discussion and Future Work

We were asked to design a system where the game environments, characters, and overarching story could be procedurally generated for increased replayability. Given these requirements, we were able to model a system architecture that was able to meet their needs.

The terrain generator component of the system was developed using a Generate and Test constrained, CFG-based approach based on Tracery's grammar system. For the High-Level History Generation and the Story Selection system a logic programming approach was selected. For the Narrative Generator an authoring language, using the plot thread approach described earlier is being developed. This will allow for the design quest templates by human designers and authors, allowing them to communicate their goals to the Drama Manager.

For the Low-Level History generation, we will be continuing development using the in-house logic programming language developed by BlastBit. This generator works backwards to add explainability to the world, describing reasons for the existence of objects, or events added to the world by the PCG generators. This approach, when fully implemented will satisfy the organization's goal of allowing the player to converse with NPCs, asking questions or reasoning about the state of the world.

A limitation of the world-first model we have designed is that the story generator focuses on modifying or fitting each story template to the generated world. Flipping this model to a story-first design, where a narrative is first generated then a world is created to fit the story, may allow the authors and game designers more freedom to focus on pacing and narrative design in the story generator.

References

Adams, T., and Adams, Z. 2006. Dwarf fortress.

Azad, S.; Xu, J.; Yu, H.; and Li, B. 2017. Scheduling live interactive narratives with mixed-integer linear programming. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 13.

Black, J. B., and Wilensky, R. 1979. An evaluation of story grammars. *Cognitive science* 3(3):213–229.

Bosser, A.-G.; Cavazza, M. O.; and Champagnat, R. 2010. Linear logic for non-linear storytelling. In *19th European Conference on Artificial Intelligence*, 713–718. IOS Press.

Brevik, D. 1997. Diablo (series).

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

Dabral, C., and Martens, C. 2020. Generating explorable narrative spaces with answer set programming. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 16, 45–51.

Doran, J., and Parberry, I. 2011. A prototype quest generator based on a structural analysis of quests from four mmorpgs. In *Proceedings of the 2nd international workshop on procedural content generation in games*, 1–8.

Dormans, J., and Bakkes, S. 2011. Generating missions and spaces for adaptable play experiences. *IEEE Transactions on Computational Intelligence and AI in Games* 3(3):216–228.

Dormans, J. 2011. Level design as model transformation: a strategy for automated content generation. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, 1–8.

Evans, R., and Short, E. 2013. Versu—a simulationist storytelling system. *IEEE Transactions on Computational Intelligence and AI in Games* 6(2):113–130.

Gellel, A., and Sweetser, P. 2020. A hybrid approach to procedural generation of roguelike video game levels. In *International Conference on the Foundations of Digital Games*, 1–10.

Grinblat, J., and Bucklew, C. B. 2017. Subverting historical cause & effect: generation of mythic biographies in caves of qud. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 1–7.

Grinblat, J., and Bucklew, C. B. 2020. Warm rocks for cold lizards: Generating meaningful quests in caves of qud. In *Experimental AI in Games Workshop (AIIDE 2020)*.

Hendrikx, M.; Meijer, S.; Van Der Velden, J.; and Iosup, A. 2013. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 9(1):1–22.

Horswill, I. D. 2018. Catsat: A practical, embedded, sat language for runtime pcg. In *Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Martens, C.; Ferreira, J. F.; Bosser, A.-G.; and Cavazza, M. 2014. Generative story worlds as linear logic programs. In *Seventh Intelligent Narrative Technologies Workshop*.

Nelson, M. J., and Smith, A. M. 2016. Asp with applications to mazes and levels. In *Procedural Content Generation in Games*. Springer. 143–157.

Persson, M., and Bergensten, J. 2011. Minecraft.

Riedl, M. O., and Bulitko, V. 2013. Interactive narrative: An intelligent systems approach. *Ai Magazine* 34(1):67–67.

Riedl, M.; Li, B.; Ai, H.; and Ram, A. 2011. Robust and authorable multiplayer storytelling experiences. In *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*.

Smith, G.; Gan, E.; Othenin-Girard, A.; and Whitehead, J. 2011. Pcg-based game design: enabling new play experiences through procedural content generation. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, 1–4.

Togelius, J.; Yannakakis, G. N.; Stanley, K. O.; and Browne, C. 2011. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games* 3(3):172–186.

Van Der Linden, R.; Lopes, R.; and Bidarra, R. 2013. Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games* 6(1):78–89.

Zhu, J.; Liapis, A.; Risi, S.; Bidarra, R.; and Youngblood, G. M. 2018. Explainable ai for designers: A human-centered perspective on mixed-initiative co-creation. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8. IEEE.