A Genre-Specific Game Description Language for Game Boy RPGs

Tamara Duplantis, Isaac Karth, Max Kreminski, Adam M. Smith, Michael Mateas
University of California, Santa Cruz
Santa Cruz, California
{tduplant, ikarth, mkremins, amsmith, mmateas}@ucsc.edu

***Abstract*—Existing game description languages (GDLs) aspire to generality, but their focus on the specification of low-level mechanics leaves game generators that target these GDLs in the awkward position of having to invent combinations of mechanics that work well together from scratch. As a result, many existing game generators are good at producing games that contain novel and surprising combinations of mechanics, but bad at generating games that are readily interpretable by players as cultural artifacts. To address this problem, we introduce the concept of a *genre-specific game description language* (GSGDL): a game description language that deliberately encodes assumptions about a particular genre of games as a cultural form. As a proof of concept, we demonstrate the use of an internal representation of game structure used by the game creation tool GB Studio as a GSGDL for top-down 2D roleplaying games targeting the Game Boy. The use of this GSGDL gives us leverage to rapidly iterate on game generation features targeting a specific game genre and platform; to work with an existing toolchain that offers graphical editing, code generation, and automated playtesting; and to more readily generate games that are interpretable by players as examples of a particular cultural form.**

I. Introduction

Game generation—the development of computational systems that can generate entire videogames—is difficult because it requires the *orchestration* of many different facets of game design, including mechanics, level design, visual art assets, narrative, and audio [1]. In successful games, these facets work together in a complementary fashion to produce a coherent play experience. In unsuccessful games, however, these facets fall out of alignment with one another—resulting in games that feel incoherent, uninterpretable, or even (in the worst case) unrecognizable as games, dispelling the illusion of an internally consistent game world.

Past approaches to game generation have often made use of *game description languages* (GDLs), including VGDL [2] and Cygnus [3], as generation targets. The use of GDLs simplifies the process of generating executable games by allowing game generators to produce high-level specifications of gameplay, rather than low-level procedural code in a general-purpose programming language. However, though these existing GDLs permit the expression of a wide range of possible game mechanics, the challenge of imbuing these novel mechanics with reliably player-interpretable meaning—including through the coherent integration of mechanics with other facets of videogames as a cultural form, such as narrative, level design, visual art assets, and sound—remains open. As a result, game generation systems targeting existing GDLs tend to be good at generating novel and surprising game *rulesets*, but bad at reliably generating games that are *interpretable* from the perspective of a human player [4].

At the same time, non-academic game development communities have developed a number of tools that aim to ease the difficulty of game development. These tools are often meant to be approachable to artists, designers, and other users who are not primarily programmers by trade, and they tend to exchange the fine-grained control of a general-purpose programming language for improved authorial leverage in creating a specific type of game. By deliberately encoding assumptions about games as a cultural form—including assumptions about game genre—these tools assist their users in creating artifacts that are *interpretable as games* with a minimum of effort.

GB Studio[1]—a GUI-based game creation tool targeting Game Boy role-playing games—is one such tool. Its internal model of game structure represents the cumulative effort of a hobbyist game development community to understand and reify the essential features of Game Boy RPGs as a genre. As developers in the community find themselves wanting to include certain features in the games they are trying to create, community discussions arise about how these features could be supported by the tool, and popularly demanded features (such as the ability to trigger a screenshake effect from a script, or to create simple turn-based battle scenes) are rolled into GB Studio or implemented as project templates that users can import into their own projects.

The resulting model of game structure is *holistic*: rather than focusing on the design of mechanics or instantial assets [5] alone, GB Studio presents users with a conceptual framework that relates assets to mechanics (and vice versa) in specific, well-defined ways, enabling users to easily construct games that creatively interleave different facets of game design while remaining interpretable to players. Essentially, GB Studio reifies the conventional configurations of operational logics [6] that characterize Game Boy RPGs as a cultural form. Users can rely on these proven proceduralist meaning-making strategies to ensure their games are interpretable to players at a high level, while improvising new settings, characters, stories,

978-1-6654-3886-5/21/$31.00 ©2021 IEEE

[1]https://www.gbstudio.dev/

puzzles, scene-specific mechanics, battle systems, inventory systems, and other facets of game design for each new game they create.

Can GB Studio guide game generators toward the creation of coherent, interpretable games, just as it guides its human users? We believe it can. To that end, we investigate the implications of treating GB Studio's internal JSON representation of game structure (the GBS format) as a machine-writable *genre-specific game description language* (GSGDL) for Game Boy RPGs. Our primary contributions include:

- The concept of genre-specific game description languages (GSGDLs): GDLs that deliberately encode genre-conventional configurations of operational logics to ease generation of player-interpretable games.
- GBS, a "found" GSGDL for Game Boy RPGs.
- Baseline approaches to game generation and machine playtesting targeting GBS.

Though formal evaluation of the generated games is left to future work, some games generated by the initial version of our system are available to play online[2] via in-browser emulation of the Game Boy platform.

II. Related Work

A. Game Description Languages for Game Generation

The Stanford GDL [7], often known simply as GDL, is an early game description language that permits the specification of rulesets for two-player perfect-information competitive turn-taking boardgames involving the placement of pieces on a grid (such as chess, checkers, and Go). GDL was originally intended to be used in the development of automated game-playing systems, enabling developers of game-playing systems to easily evaluate their approach against a large number of different game rulesets. It later served as an inspiration for a number of boardgame description languages, including the Ludi GDL—a narrower and more focused boardgame description language. Ludi was used to generate boardgames that human players found compelling, including multiple commercially published games [8]. Other GDLs for board games include the Extensible Graphical Game Generator's `.egg` format [9] and the Zillions of Games `.zrf` format[3].

VGDL [2] is a game description language inspired by GDL, but targeting the automated playing and generation of 2D arcade videogames rather than boardgames [10]. The game AI research toolkit GVGAI [11] uses VGDL as a shared representation of game structure between automated game-playing and game generation systems. VGDL can represent only a limited subset of arcade game rulesets; for instance, VGDL hardcodes a fixed number of opaque control schemes rather than allowing for the invention of new control schemes from scratch, and permits game entities to interact with one another only through collisions, rather than (for instance) through quantitative resources.

[2]https://isaackarth.com/games/rom_gen_test_5/

[3]https://www.zillions-of-games.com/

The Cygnus game description language [3] was developed in response to VGDL's limitations. Cygnus permits game mechanics to be defined as arbitrary compositions of triggers (e.g. control events, entity collisions, and resource thresholds being met or surpassed) and effects (e.g. setting entity movement properties such as rotation or acceleration, adding or removing entities, and altering resource values). As a result, Cygnus is capable of expressing a strict superset of the games expressible in VGDL. Cygnus powers the abstract game generator Gemini [3], which has been used to generate small arcade games included in the narrative game *Emma's Journey* [12] as thematically appropriate procedural accompaniment to a text-driven narrative. It also powers the Gemini-based mixed-initiative game creation tool Germinate [13].

PuzzleScript [14], a GDL originally developed in a non-academic context for use by human creators of top-down 2D tile-based block-sliding puzzle games, has been used as a target for both level generation [15] and full game generation including the invention of novel mechanics [16]. Additionally, the Inform 7 [17] and Ceptre [18] languages could both be viewed as GDLs for narrative games. Inform 7 has been used as a target for text adventure game generation as part of the TextWorld framework [19].

B. Other Approaches to Game Generation

Not all game generators make use of explicitly defined GDLs. One early approach to videogame generation by Nelson and Mateas [20] generates WarioWare-esque microgames by combining a few sets of stock mechanics with entity movement behaviors constrained by common-sense knowledge of object types from the ConceptNet [21] and WordNet [22] databases. This approach served as an inspiration for the later Game-O-Matic [23] and Gemini systems.

Another early approach by Togelius and Schmidhuber [24] evolves arcade games within a search space defined by an internal representation of game rules. However, this representation is not externally exposed as a GDL. Smith and Mateas's *Variations Forever* system [25] similarly makes use of an internal-only representation of game rule structure for minigame generation, but instead uses answer set programming to search the space for viable rulesets. The Gamika system defines another space of minigames, but turns the task of exploring the space over to a human user via casual creator interaction patterns in Wevva [26] and other apps.

The evolutionary approach to game generation was carried forward by Cook's ANGELINA series of systems [27], which have used a variety of internal representations of game structure. More recent versions of ANGELINA [28] have experimented with recontextualizing game generation as a continuous [29] and culturally engaged process, with ANGELINA querying its Twitter followers for common-sense knowledge about how game entities might be expected to relate to one another; theming its games after news articles it consumes; and entering its games into game jams.

WikiMystery [30], another experiment with culturally engaged videogame generation, uses real-world data (specifically

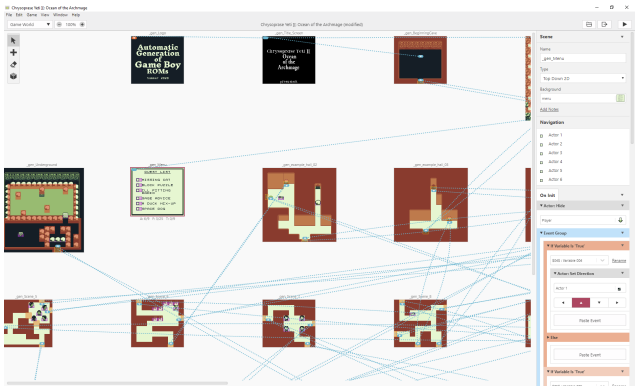

Fig. 1. The GB Studio 1.2.1 interface, displaying the project file for the generated game *Crysopraise Yeti][: Ocean of the Archmage*. Scenes are shown as rectangles in the project view. Entity properties, including scripts, are accessed through the bar on the right.

Wikipedia data about networks of well-known historical figures) to generate murder mystery adventure games that evoke a specific cultural context.

Machine learning approaches to the acquisition of game design knowledge for game generation have recently come into use [31]. Guzdial and Riedl [32] combine these approaches with conceptual expansion to invent novel gameplay through the recombination of mechanics learned from existing games.

Cook has recently argued [33] that game generation tools intended to fit into human game development workflows could benefit from targeting general-purpose programming languages directly, rather than targeting a narrower GDL. Such tools would need to both read human-authored code (perhaps structured in a way that makes this code more amenable to automatic analysis) and to emit human-readable code.

III. Problem Space Definition

We want to generate Game Boy role-playing games that center on the exploration of a navigable graphical environment, consisting of a set of scenes linked to one another by collision-triggered or interaction-triggered doorways and containing interactable entities of the types that are common to the genre (including non-player characters, readable wall signs, toggleable switches, and so on). NPCs and other entities should be able to present players with textual dialogue or descriptions on interaction, and players should be able to make choices among several fixed options within dialogue sequences. Additionally, a scripted narrative throughline should be presented to the player as they progress through a sequence of key interactions. Examples of the kinds of explorable overworlds that we would like to capture include those found in *Pokémon Red* [34], *The Legend of Zelda: Link's Awakening* [35], *Final Fantasy Legend II* [36], and *Great Greed* [37].

Neither VGDL nor Cygnus seems appropriate to encoding games of this type. VGDL allows the specification of level layouts, sprites, and how sprites behave when they collide with each other [38], while Cygnus generalizes VGDL in several ways, including through the incorporation of *resource logics* that allow entity behaviors to depend on and alter the values of quantitative resources [3]. However, neither language offers any built-in means of presenting dialogue boxes, menus, or characters emoting. Transitions between linked scenes in a larger persistent world are outside the vocabulary of both languages. Additionally, RPGs tend to tightly interleave narrative, level design, sprite design, and interaction: for instance, NPCs often have character-specific dialogue and movement sequences that are triggered by interacting with them, that only make sense if certain objects are present in the room, and that vary based on player input at designated interaction points. These scripted "cutscenes" are hard to express in languages developed to express arcade game mechanics, where it's generally assumed that there are a small number of distinct entity types that behave in consistent ways throughout the game, rather than many entities with highly situational behaviors, each appearing in only a few scenes. Though these constructs could perhaps be expressed as combinations of low-level operations available in existing GDLs—for instance by defining scene transitions as collision-triggered Cygnus rules that remove all currently active entities and immediately place a new set of entities to construct a new scene—the expression of these basic player-facing concepts as elaborate compositions of lower-level operations inhibits the ability of game generators to reason about these concepts directly.

One way to frame these difficulties is through the lens of *operational logics* (OLs). The theoretical framework of operational logics attempts to describe how videogames create meaning by connecting computational processes to representational strategies, forming low-level operational logics that can be composed to create playable models. Mateas and Wardrip-Fruin [39] provided the first full characterization of OLs; Osborn, Wardrip-Fruin and Mateas [6] later refined the notion of OLs by providing a catalog of common foundational logics and an account of how OLs compose. For our purposes, OLs are useful because they describe the relationship between game mechanics and representational strategies in a way that permits us to capture the foundational elements of Game Boy RPGs as a cultural form. Sets of formal characteristics that are common to recognizable categories of games might in some contexts be referred to as game *genres*—a term that has been critiqued for its vagueness [40]. Therefore, we want a more precise definition of exactly what we mean when we refer to Game Boy RPGs as a genre. In the remainder of this paper, we use the term "genre" to refer to a *conventional configuration of operational logics* in relation to one another. This sense of "genre" is roughly equivalent to Lessard's "high-level design pattern formations" [41], but formalized in terms of OLs.

From the OLs perspective, Game Boy RPGs present a navigable overworld consisting of a number of scenes, connected to one another by linking logics. Collision volumes attached to graphical representations of doorways or corridors trigger the operation of these linking logics when activated, transporting the player character from one scene to the next. Interaction with NPCs and other entities relies on a conventional playable model of interaction proxemics, consisting of

a structural synthesis of collision logics (used to determine whether the player character is touching another entity), entity-state logics (used to determine whether the player character is facing the contacted entity), and control logics (used to trigger interactions when the player presses a specific button). Within each interaction, a progression logic is used to trigger an ordered sequence of dialogue boxes, character emotes, character movements, and other changes to the game state. Additionally, within these sequences, players may be presented with menus backed by a selection logic, enabling them to make choices that impact the outcome of the sequence as a whole. Finally, persistence and progression logics are used to craft a high-level narrative arc by causing the game world to evolve in a scripted way as the player completes key interactions.

This perspective helps characterize why existing GDLs are insufficient for our purposes. As Beaupre et al have pointed out [42], the operational logics expressed by games in the popular GVGAI framework for "general" game playing and generation are in fact quite limited. This is due in part to the GVGAI framework's reliance on VGDL, which was designed to express the mechanics of arcade games and consequently does not provide language features relevant to, for example, the expression of the narrative facet of adventure games. Cygnus makes similar arcade game-specific assumptions about desired OLs. Altogether, it is this mismatch between the configurations of OLs presumed by existing GDLs and those conventional to Game Boy RPGs as a genre that prompted us to seek out an alternative language: one that better supported the genre conventions we wanted our generator to target.

IV. GBS: A Genre-Specific GDL

GB Studio is a GUI-based game creation tool that targets the Nintendo Game Boy. Our Game Boy RPG generation pipeline uses GB Studio's internal JSON representation of game structure—which we call GBS—as a machine-writable *genre-specific game description language* (GSGDL) (Fig. 2). A GSGDL, by analogy to domain-specific programming languages, is a GDL that sacrifices generality for ease of use in generating a particular kind of game. GBS specifically trades generality for suitability to Game Boy RPG generation by directly encoding key features of the Game Boy RPG as a cultural form—including interactable NPCs with scripted interaction sequences, an overworld consisting of spatially linked scenes, and interactive menus—as language features.

GBS games are structured around five major types of entity.

Scenes are rectangular virtual spaces within which the player and actors may exist. Every scene has a background consisting of 8-by-8 pixel tiles and a collision map defining which tiles block movement by the player and actors. Scenes are linked together by scripts that move the player to a specific scene when run, and they can be pushed onto and popped off of a scene stack. This stack mechanism essentially allows scenes to contain child scenes, which the player can enter and return from without losing the state of the parent scene.

The player is an avatar controlled by the human player, represented by a specific sprite or sprite set. Every scene provides a specific start location at which the player will appear when the scene is entered. The player can then be moved around the scene via the directional input buttons and made to interact with actors via the interact button (conventionally the A button, or Z in an emulator using a keyboard).

```
{
  "scenes": [{
    "id": "SebsHouse",
    "backgroundId": "...",
    "width": 20, "height": 18,
    "collisions": [255, 255, 25, 248...],
    "actors": [{
      "spriteSheetId": "...", "animSpeed": "3",
      "movementType": "randomFace",
      "moveSpeed": "1",
      "direction": "down", "x": 8, "y": 7,
      "script": [
        {"command": "EVENT_TEXT",
         "args": {"text": "Have you seen my keys?",
                  "avatarId": "..."}},
        {"command": "EVENT_MENU",
         "args": {"layout": "dialogue",
                  "variable": "0",
                  "option1": "Yes",
                  "option2": "No"}}
      ]
    }],
    "triggers": [{
      "x": 17, "y": 0, "width": 2, "height": 1,
      "script": [
        {"command": "EVENT_SWITCH_SCENE",
         "args": {"sceneId": "StansHouse",
                  "x": 1, "y": 16,
                  "direction": "up",
                  "fadeSpeed": "2"}}
      ]
    }]
  }]
}
```

Fig. 2. A lightly edited example of GBS syntax specifying a game that contains a single scene, a single actor with a dialogue script triggered by interaction, and a single trigger that takes the player to an adjacent scene when entered.

Actors are interactable entities represented by a specific sprite or sprite set, which can move around within a scene. When the player faces an actor and presses the interact button, the script attached to the actor (if any) is executed. Four basic movement types can be applied to actors: *static*, or fixed in place; *face interaction*, like static but turning to face the player when interacted with; *random rotation*, or periodically changing facing direction at random; or *random movement*, i.e. periodically changing direction and moving around the scene. All but the static movement type are intended to replicate common forms of NPC behavior in Game Boy RPGs. Additionally, more complex movements can be achieved through scripts.

Triggers are areas that run a script when entered by the player. They consist of one or more 8px by 8px tiles, and are often used to link scenes together and to trigger cutscenes.

Scripts are sequences of commands that run when a specific interaction takes place. They can be attached to scenes (running when the player enters the scene), to actors (running when the player interacts with the actor), or to triggers (running

when the player enters the trigger). Additionally, scripts can set other scripts to be run whenever a specific console button is pressed or on a repeating interval, and they can trigger scripts attached to actors as though the player interacted with the actor in question. Scripts offer both branching and looping control flow constructs and commands for manipulating values stored in variables, allowing them to function as a general-purpose programming language. However, they also offer RPG-specific commands, such as displaying text in a dialogue box (with a parameter to control the typeout speed) or displaying a menu that blocks execution until the player selects one of several options. Accordingly, scripts are often simple and unconditional, consisting of a single command that displays several lines of text in sequence, for instance to simulate the player reading a sign or talking to an NPC.

In addition to these mechanical entity types, GBS also defines a set of fixed roles for assets (including visual and audio assets) that, together with the limitations of the Game Boy as a target platform, impose a degree of aesthetic consistency on generated games. Sprites designated as emotes will appear above the target actor when triggered by an appropriate script command; UI elements, such as the selection cursor and text box borders, can be visually redesigned while continuing to play a fixed interaction role; and spritesheets for actors that are capable of rotation must provide variant sprites for the four cardinal directions in which an actor can be facing. These representational conventions help support the mechanical conventions that the GBS language imposes.

Together, the genre-specific mechanical and communicative features of the GBS GDL can be interpreted as reifying the distinctive configurations of operational logics that characterize Game Boy RPGs as a genre. Scenes are connected spatially by linking logics, whose operations are triggered by collision logics or scripted interaction sequences; dialogue boxes, emotes, character movements, and menus can be triggered by single script commands; and so on. Basing design on a known-good high-level configuration of operational logics can be helpful to a game generator in the same way that deciding to work within a specific genre can be helpful to a human designer: the genre structure provides a set of recognizable conventions within which to work, closing off many possible alternative configurations of logics but ensuring that a coherent assemblage of mechanics does not have to be discovered through painstaking trial and error. Meanwhile, though the scripting capabilities of the GBS language permit the specification of novel mechanics, the affordances of the language guide both human authors and game generators toward scripts that mostly follow the conventions of the Game Boy RPG.

V. Game Generation via GB Studio

As a proof of concept for our approach, we built a simple constructive game generator that targets GBS (Fig. 3). Further description of the generated games is available in our demonstration paper about the generator itself [44]. The success of this generator at producing interpretable games, despite its relatively simple architecture in comparison to other game

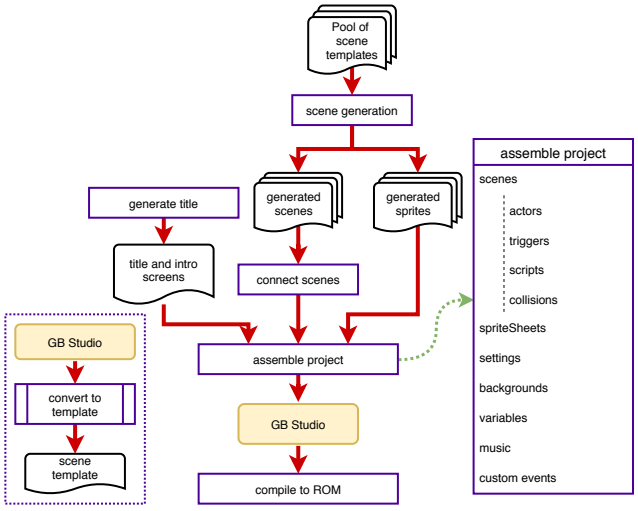

Fig. 3. The generative pipeline in version 1.0 of the generator. The generator starts with a user-supplied pool of scene templates (as Python functions) and executes the functions, creating the data for the scenes and associated assets. The generated scenes are then connected by creating bi-directional links between scenes. As a separate process, the title of the game (generated via Tracery [43]) is used to generate the title screen. The generated data is assembled into a project data structure, which is written to disk and then read into GB Studio, which compiles the project into a ROM. The scene template functions can be hand-written, or can be extracted from other GB Studio project files with a Python script.

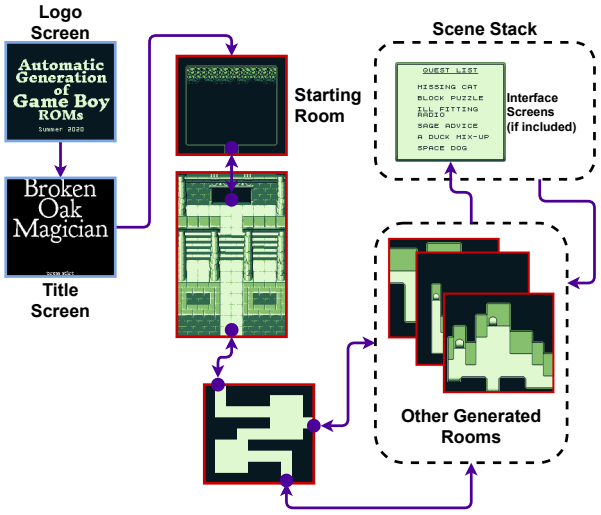

Fig. 4. Diagram of the basic layout of *Broken Oak Magician*, a generated game. Execution begins with the intro and title screens (top left), from which the player can start a new game or load a saved game. New games start in the scene the generator has designated as the starting room. The scenes are connected via scripts on trigger volumes, creating bidirectional links. Additional interface scenes and modes (menus, combat, etc.) can be accessed using GB Studio's scene stack, but *Broken Oak Magician* doesn't use these more advanced scene-switching features.

generators, demonstrates the benefits of using a GSGDL as a target for game generation.

Generation is based on a template-and-mixin model. The generator first selects several *region modules* (e.g., the "forest", "sewer", and "temple" regions) from a list of available regions. Then, for each of the selected regions, it calls the region's `catalog()` function, which returns a sampled subset of appropriate *scene templates* from the region's template library. Scene templates are Python functions that generate high-level scene geometry, including backgrounds and collision maps. Once the scene templates are selected, each template is *invoked* to generate a single scene.

The template library currently contains 70 templates, most of which were hand-authored using the graphical GB Studio interface. To templatize these hand-authored scenes, we use a Python script that accepts a GBS file as input, analyzes the scenes present within it, and generates a Python function for each scene that is capable of producing slight variations on that scene's basic geometry when invoked—essentially conducting a simple generativist reading [45] of the human-crafted example scenes. These automatically-templatized functions can be further hand-modified to increase their expressive range. A few templates, however, are defined algorithmically, and generate a completely new scene geometry (such as a procedural maze) each time they are invoked.

All scene templates include one or more marked connection points (such as doors, ladders, and offscreen-pointing corridors), tagged with a list of scene types to which this scene may be connected via this connection point. Connection points with matching tags are randomly connected to one another, forming one sub-graph of interconnected scenes per region. For each connection, a trigger is placed in the connected scenes at each end of the connection. Attached to this trigger is a script containing a single `EVENT_SWITCH_SCENE` command that will move the player to the appropriate location within the connected scene.

Scenes that connect one region to another region are treated specially, and used to implement both visually smooth transitions between regions and a succession of locked gates that limit progression through the game world. Traversing a connection from one region to the next first checks to determine whether a specific flag variable, corresponding to the acquisition of a particular key item, has been set. Key items are placed within certain scene templates, and exactly one scene containing a key item is generated per region; key items can be collected by interacting with a representation of the item itself, with a particular NPC who carries the item, or by entering a certain trigger to "pick up" a key item from the ground. This limits progression from one region to the next until the player collects the key item within that region. Because scene-to-scene links are implemented as scripts, the script for a particular link can easily be modified to add narrative or progression elements, including additional dialogue or a conditional "lock" that determines whether the link can be traversed.

We evaluate the traversability of the scene graph as a whole

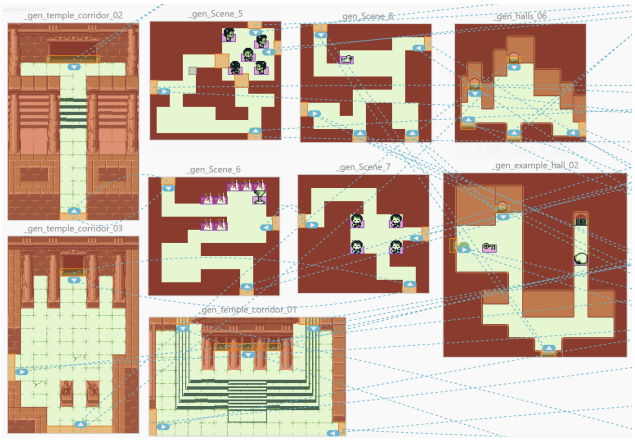

Fig. 5. A subset of the scenes placed in the generated game *Broken Oak Magician*, demonstrating how the templates are turned into finished scenes with trigger-based connections between the scenes. The layout of scenes in the project view is arbitrary. At places where a scene can connect to another scene, its scene generation function adds a trigger with an `EVENT_SWITCH_SCENE` script command that points at the scene (blue boxes with arrows). A later pass selects pairs of self-referencing triggers and swaps destinations, creating bi-directional links (indicated with blue dotted lines). Collision areas, derived from the original template scene, are highlighted in amber.

via a depth-first search to ensure that a path can be found from the beginning of the game world to the end. Then we augment scenes by placing randomly chosen mixins at certain marked-up decoration points. Mixins include wandering animals, humorous road signs, and NPCs; for instance, a mentor figure who tells the player about their quest can be placed placed in one of the first few rooms of the game. Dialogue for NPCs is generated via a Tracery [43] grammar containing template variables, which are then swapped out for game-specific strings, such as the name of the game's primary MacGuffin. The art assets for mixin NPCs are randomly drawn from a pool of possible NPC sprites. Additionally, another templated Tracery grammar is used to generate the game's title, which is typeset and rendered out as an image for use as a title screen.

Once the generated game design is finalized, it is sent to GB Studio for compilation. To write our internal Python representation of GBS to a `.gbsproj` file that GB Studio can consume, we first reify our generator's internal entity IDs into GUIDs (globally unique identifiers) for GB Studio, as GB Studio uses GUIDs for cross-referencing between scripts and entities. We create a folder for image and audio assets, so that GB Studio knows where to locate them, and copy all assets referenced by the game (including scene backgrounds produced by the scene generation step of our pipeline) to the asset folder. The Python game data structure is then translated to JSON for export to a `.gbsproj` file.

Finally, we load the generated `.gbsproj` file into GB Studio and invoke its compilation pipeline to produce a playable ROM. GB Studio's frontend is a cross-platform Electron application, while its backend is built on top of the Game Boy Developer's Kit (GBDK), an open source set of tools and libraries. GBDK is mostly implemented in C and uses

the Small Device C Compiler (sdcc) to compile ROMs that will run on the Game Boy Z80 architecture used by the Game Boy hardware. The resulting ROM can then be loaded into a Game Boy cartridge using a cartridge programmer or run in the browser via emulator. This is an advantage of the maturity of the Game Boy emulation ecosystem, and one way our research benefits from our focus on a platform with extensive in-the-wild support from developers.

VI. Machine Playtesting with Go-Explore

The capacity for machine playtesting of generated games is a desirable feature for a game generation pipeline. Game generators can make internal use of machine playtesting to evaluate and improve their unfinished game designs, for instance as a means of determining fitness in evolutionary approaches to game generation. External evaluators can use machine playtesting on a generator's final output to gauge the generator's success. And human users of mixed-initiative game creation tools can use machine playtesting to identify problems with their game designs. The GVGAI framework uses VGDL as a target for machine playtesting to enable the machine playtesting of generated rulesets; however, we do not need to implement a comparable GBS-specific machine playtesting infrastructure in order to enable machine playtesting of our games. Instead, we take advantage of the existing ecosystem of machine playtesting tools for compiled Game Boy ROMs.

We use the Open AI Gym Retro API [46] to step through the execution of a game frame-by-frame, providing different controller inputs over time (as one might expect of the forward execution model for games in the VGDL representation). We apply the Go-Explore algorithm [47] (one of many available automated game exploration methods [48]) to find samples of diverse pathways through the game's state space. By running this exploration process for some duration (e.g. for one minute of wall-clock, during which about one hour of virtual gameplay is simulated), we can render short videos (e.g. 10 seconds of gameplay) demonstrating samples of gameplay in which the agent appears to optimize for exploration. Even though the exploration algorithm makes use of random button pressing as the main input strategy, the algorithm records the sequences which successfully made progress towards touching previous unseen gamestates. Chains of such sequences demonstrate making rapid progress through the different screens of the game, including by successfully solving simple generated lock-and-key puzzles that act as gates to progression.

Though this approach to machine playtesting is well-suited to evaluating the reachability of areas and content in generated games, it is not as well-suited to evaluating other aesthetic or experiential aspects of gameplay. Future efforts to develop machine playtesting tools for GBS games may leverage the GBS description of game structure to augment ROM-based playtesting systems with further understanding of game structure—perhaps enabling these tools to assess features of game design such as puzzle difficulty or narrative pacing. Additionally, machine playtesting of Game Boy RPGs created by a simple constructive generator (such as the generator presented here) may be used to establish a baseline of comparison for future approaches to the generation of GBS games.

VII. Discussion and Conclusion

We have presented GBS: a novel, machine-writable genre-specific game description language (GSGDL) for Game Boy RPGs. This language was extracted from the internal representation of game structure used by the GUI-based game creation tool GB Studio. Unlike previous game description languages, GBS is well-suited to the expression of multi-layered RPG structures, enabling even a relatively naïve game generator to produce games that are recognizable by players as examples of the Game Boy RPG as a cultural form. To demonstrate the expressive capabilities of the GBS language, we have presented a simple constructive game generator targeting GBS. We have also shown that an existing machine playtesting system for Game Boy games can be straightforwardly applied to the ROMs produced through compilation of GBS games.

For game generation researchers, the potential benefits of targeting a game platform and genre supported by an active community of practice are many. Channels for distributing GB Studio games to the public already exist; for instance, many such games have been made available through the itch.io indie game hosting platform, which provides tagging features to make games discoverable to potential players and allows for the hosting of a web emulator capable of playing Game Boy ROMs in the browser. Because approachable development tools for Game Boy RPGs already exist, there is immediate potential for educational impact in the deployment of undergraduate-facing design-assistance tools that use the GDL as a game sketching language. Even in its raw JSON form, the GBS GDL is approachable enough that several high-school-aged research interns were successfully able to hand-write GBS templates to add to our generator. And finally, in attempting to develop more sophisticated game generators targeting GBS, we can draw on a sizable existing corpus of premade games (both GB Studio projects and Game Boy ROMs) as training data. All of these advantages increase the potential that our future work will be impactful due to its engagement with a living game development ecosystem.

We have argued that genre-specific game description languages, which encode genre-conventional configurations of operational logics, can help to address the problem of orchestration in game generation. One path forward for research in game generation, therefore, involves the formalization of game genre from an OLs perspective, followed by the development of GSGDLs that accurately capture the conventional configurations of OLs that characterize genres of interest. This future work, too, stands to benefit from engagement with existing communities of practice. Because GBS was created by a particular community of game development practice to enable creation of the specific kinds of games they sought to create, it can be viewed as a communal repository of game design knowledge pertaining to a particular target platform and genre. We believe that it may be fruitful to learn from

other existing hobbyist game development communities in the process of attempting to develop new GSGDLs.

More generally, it is our hope that future work in game generation will more actively engage with videogames as a multifaceted cultural form. Some existing game generators, such as ANGELINA, have attempted to generate culturally resonant games, but many important creative facets of games—including the narrative dimension of role-playing games—still tend to be sidelined by a perspective on game generation that treats automated game design primarily as a problem of inventing novel rulesets. Videogames create meaning not through rules alone, but through the creative interleaving of abstract computational processes with representational strategies. Consequently, we believe that our automated game design systems—including the languages they target—must embody a holistic perspective on games as cultural artifacts if they are to generate games that players find meaningful.

ACKNOWLEDGMENT

We would like to thank Science Internship Program interns Sachita Kashyap, Vijaya Kukutla, Aaron Lo, Anika Mittal, and Harvin Park for their contributions to the game generator. Thanks also to the anonymous reviewers for their thoughtful feedback on this and earlier versions of this paper.

REFERENCES

[1] A. Liapis, G. N. Yannakakis, M. J. Nelson, M. Preuss, and R. Bidarra, "Orchestrating game generation," *IEEE TOG*, vol. 11, no. 1, 2018.

[2] T. Schaul, "A video game description language for model-based or interactive learning," in *Proc. IEEE CIG*, 2013.

[3] A. Summerville, C. Martens, B. Samuel, J. Osborn, N. Wardrip-Fruin, and M. Mateas, "Gemini: bidirectional generation and analysis of games via ASP," in *Proc. AIIDE*, 2018.

[4] J. C. Osborn, M. Dickinson, B. Anderson, A. Summerville, J. Denner, D. Torres, N. Wardrip-Fruin, and M. Mateas, "Is your game generator working? Evaluating Gemini, an intentional generator," in *Proc. AIIDE*, 2019.

[5] M. Treanor, B. Schweizer, I. Bogost, and M. Mateas, "The micro-rhetorics of Game-O-Matic," in *Proc. FDG*, 2012.

[6] J. C. Osborn, N. Wardrip-Fruin, and M. Mateas, "Refining operational logics," in *Proc. FDG*, 2017.

[7] M. Genesereth, N. Love, and B. Pell, "General game playing: Overview of the AAAI competition," *AI Magazine*, vol. 26, no. 2, 2005.

[8] C. Browne and F. Maire, "Evolutionary game design," *IEEE TCIAIG*, vol. 2, no. 1, 2010.

[9] J. Orwant, "EGGG: the extensible graphical game generator," Ph.D. dissertation, Massachusetts Institute of Technology, 2000.

[10] M. Ebner, J. Levine, S. M. Lucas, T. Schaul, T. Thompson, and J. Togelius, "Towards a video game description language." Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

[11] D. Perez-Liebana, J. Liu, A. Khalifa, R. D. Gaina, J. Togelius, and S. M. Lucas, "General video game AI: A multitrack framework for evaluating agents, games, and content generation algorithms," *IEEE TOG*, 2019.

[12] J. Garbe, M. Kreminski, B. Samuel, N. Wardrip-Fruin, and M. Mateas, "StoryAssembler: an engine for generating dynamic choice-driven narratives," in *Proc. FDG*, 2019.

[13] M. Kreminski, M. Dickinson, J. Osborn, A. Summerville, M. Mateas, and N. Wardrip-Fruin, "Germinate: A mixed-initiative casual creator for rhetorical games," in *Proc. AIIDE*, 2020.

[14] S. Lavelle, "Puzzlescript," http://puzzlescript.net, 2013.

[15] A. Khalifa and M. Fayek, "Automatic puzzle level generation: A general approach using a description language," in *Proc. CCGW*, 2015.

[16] C.-U. Lim and D. F. Harrell, "An approach to general videogame evaluation and automatic generation using a description language," in *Proc. IEEE CIG*, 2014.

[17] G. Nelson, "Inform 7," http://inform7.com, 2019.

[18] C. Martens, "Ceptre: A language for modeling generative interactive systems," in *Proc. AIIDE*, 2015.

[19] M.-A. Côté, A. Kádár, X. Yuan, Q. Kybartas, T. Barnes, E. Fine, J. Moore, R. Y. Tao, M. Hausknecht, L. E. Asri, M. Adada, W. Tay, and A. Trischler, "TextWorld: A learning environment for text-based games," *CoRR*, vol. abs/1806.11532, 2018.

[20] M. J. Nelson and M. Mateas, "Towards automated game design," in *Proc. AI*IA*, 2007.

[21] H. Liu and P. Singh, "ConceptNet—a practical commonsense reasoning tool-kit," *BT Technology Journal*, vol. 22, no. 4, 2004.

[22] P. Singh, "The public acquisition of commonsense knowledge," in *Proc. AAAI Spring Symposium: Acquiring (and Using) Linguistic (and World) Knowledge for Information Access*, 2002.

[23] M. Treanor, B. Blackford, M. Mateas, and I. Bogost, "Game-O-Matic: Generating videogames that represent ideas," in *Proc. PCG*, 2012.

[24] J. Togelius and J. Schmidhuber, "An experiment in automatic game design," in *Proc. IEEE CIG*, 2008.

[25] A. M. Smith and M. Mateas, "Variations Forever: Flexibly generating rulesets from a sculptable design space of mini-games," in *Proc. IEEE CIG*, 2010.

[26] E. Powley, M. Nelson, S. Gaudl, S. Colton, B. P. Ferrer, R. Saunders, P. Ivey, and M. Cook, "Wevva: Democratising game design," in *Proc. AIIDE*, 2017.

[27] M. Cook, S. Colton, and J. Gow, "The ANGELINA videogame design system—part I," *IEEE TCIAIG*, vol. 9, no. 2, 2016.

[28] ——, "The ANGELINA videogame design system—part II," *IEEE TCIAIG*, vol. 9, no. 3, 2016.

[29] M. Cook and S. Colton, "Redesigning computationally creative systems for continuous creation," in *Proc. ICCC*, 2018.

[30] G. A. B. Barros, M. Green, A. Liapis, and J. Togelius, "Who killed Albert Einstein? From open data to murder mystery games," *IEEE TOG*, 2019.

[31] J. C. Osborn, A. Summerville, and M. Mateas, "Automated game design learning," in *Proc. IEEE CIG*, 2017.

[32] M. Guzdial and M. Riedl, "Automated game design via conceptual expansion," in *Proc. AIIDE*, 2018.

[33] M. Cook, "Software engineering for automated game design," in *Proc. IEEE CoG*, 2020.

[34] Game Freak, "*Pocket Monsters: Red*," [Game Boy cartridge], 1996.

[35] Nintendo, "*The Legend of Zelda: Link's Awakening*," [Game Boy cartridge], 1993.

[36] Square, "*Final Fantasy Legend II*," [Game Boy cartridge], 1990.

[37] Namco, "*Great Greed*," [Game Boy cartridge], 1992.

[38] T. S. Nielsen, G. A. Barros, J. Togelius, and M. J. Nelson, "Towards generating arcade game rules with VGDL," in *Proc. IEEE CIG*, 2015.

[39] M. Mateas and N. Wardrip-Fruin, "Defining operational logics," in *Proc. DiGRA*, 2009.

[40] M. Treanor and M. J. Nelson, "Order-fulfillment games: An analysis of games about serving customers," in *Proc. FDG*, 2019.

[41] J. Lessard, "Game genres and high-level design pattern formations," in *Proc. FDG*, 2014.

[42] S. Beaupre, T. Wiles, S. Briggs, and G. Smith, "A design pattern approach for multi-game level generation," in *Proc. AIIDE*, 2018.

[43] K. Compton, B. Kybartas, and M. Mateas, "Tracery: an author-focused generative text tool," in *Proc. ICIDS*, 2015.

[44] I. Karth, T. Duplantis, M. Kreminski, S. Kashyap, V. Kukutla, A. Lo, A. Mittal, H. Park, and A. M. Smith, "Generating playable RPG ROMs for the Game Boy," in *Proc. FDG*, 2021.

[45] M. Kreminski, I. Karth, and N. Wardrip-Fruin, "Generators that read," in *Proc. FDG*, 2019.

[46] A. Nichol, V. Pfau, C. Hesse, O. Klimov, and J. Schulman, "Gotta learn fast: A new benchmark for generalization in RL," *arXiv preprint arXiv:1804.03720*, 2018.

[47] A. Ecoffet, J. Huizinga, J. Lehman, K. O. Stanley, and J. Clune, "Go-explore: a new approach for hard-exploration problems," *arXiv preprint arXiv:1901.10995*, 2019.

[48] Z. Zhan, B. Aytemiz, and A. M. Smith, "Taking the scenic route: Automatic exploration for videogames," in *Proc. KEG*, 2019.