GIGL: A Domain Specific Language for Procedural Content Generation with Grammatical Representations

Tiannan Chen, Stephen J. Guy
Department of Computer Science & Engineering, University of Minnesota
chen2814@umn.edu, sjguy@umn.edu

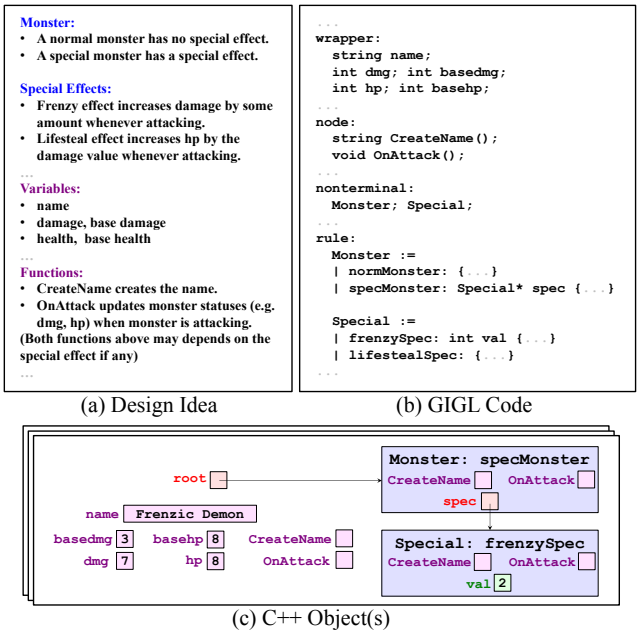
Abstract

We introduce a domain specific language for procedural content generation (PCG) called *Grammatical Item Generation Language* (GIGL). GIGL supports a compact representation of PCG with stochastic grammars where generated objects maintain grammatical structures. Advanced features in GIGL allow flexible customizations of the stochastic generation process. GIGL is designed and implemented to have direct interface with C++, in order to be capable of integration into production games. We showcase the expressiveness and flexibility of GIGL on several representative problem domains in grammatical PCG, and show that the GIGL-based implementations run as fast as comparable C++ implementation and with less code.

Introduction

Procedural content generation (PCG), algorithmic creation of content with limited or indirect user input (Togelius et al. 2011), has been adopted in commercial games such as *Diablo III* (Blizzard) and *Path of Exile* (Grinding Gear Games), and studied in many academic contexts including plants (Smith 1984), and furniture layouts (Germer and Schwarz 2009). It is also found in domains outside of entertainment, such as automatically generating code as test cases for software verification (Claessen and Hughes 2011) or randomly generating math quizzes (Tomás and Leal 2013). In general, PCG is essential when large amount of diverse content is needed, such as in creating large virtual world (Smelik et al. 2011), and creating scenes to test multi-agent simulations (Arnold and Alexander 2013).

Many interesting PCG problems are inherently hierarchical. These hierarchies can be formalized as grammars, which provide a natural way of expressing the relationship between various components. Examples include the self-similarity seen in plants, component-wise relationships that make up special items in games, or spatial relationships seen in level designs (Shaker, Togelius, and Nelson 2016). By incorporating grammars into the creation of these objects at the code level, we can achieve a compact representation of many complex objects, and can re-use the structures to improve computational efficiency (e.g. as spatial-data structures).



(a) Design Idea (b) GIGL Code

(c) C++ Object(s)

Figure 1: (a) The design idea for generating random monsters in an RPG game; (b) Key parts of the GIGL code; (c) Example C++ object(s) generated, with the state shown as after name being created and having attacked twice (function contents omitted). Color scheme: blue indicates rules to expand a nonterminal; red indicates nonterminals (pointers); green indicates terminals; purple indicates attributes, which can be variables or functions.

While grammars are useful in PCG, the difficulty of implementing grammars can be an obstacle to their adoption. On one hand, general purpose languages used in most game programming (e.g. C++) typically do not have direct support for grammars, requiring users to implement their own type systems, hierarchical structures, grammar parametrization, constraint mechanisms, etc. On the other hand, existing domain specific languages (DSLs) for grammatical PCGs are typically more narrowly targeted than general purpose languages, without support for compilation to executables, integration with existing code bases, and high-speed runtime performance, all of which are frequently needed by mod-

ern games. Our goal, then, is to close the gap between the ease of designing a grammatical PCG and the issues in implementing it. For this purpose, we introduce a DSL called *Grammatical Item Generation Language* (GIGL). As shown in Fig. 1, GIGL expresses the design idea of a PCG problem in a natural way (Fig. 1a & 1b), and can be compiled directly into C++ objects (Fig. 1c) while still maintaining the grammatical structures.

Our work here presents three main contributions:

- *Grammatical Item Generation Language (GIGL)*: We design GIGL, a compilable DSL efficiently creating grammatical PCG that can directly interface with C++ code.
- *Customizable Item Generators*: GIGL is based on a PCG formalism we refer to as *item grammars*, on top of which we provide support on flexible control over the stochastic generation of the procedural content.
- *High-Performance PCG*: GIGL is implemented to have high runtime performance (close to C++). In addition, the grammatical structures stored in the generated C++ objects can be exploited for further optimization.

It is important to note that throughout the text we use the word *item* to broadly mean any procedurally generated content, such as monsters, puzzles, mazes, plants, dungeons, etc.

Background and Related Work

Our work relates both to the use of grammars in PCG and DSL for games. We briefly survey both areas below.

Grammatical PCG

Grammar-based PCG approaches have been widely used in a variety of settings. Grammars have been used for procedural level design by formulating player action rhythms as rules (Smith et al. 2009), as representations in evolving Mario levels (Shaker et al. 2012), for generating environments for an endless-run type game (Toto and Vessio 2014), and for generating instances for MMORPG games (Merrick et al. 2013). Grammars have also been used in procedural modeling of 3D scenes (Krecklau and Kobbelt 2012), cities (Parish and Müller 2001; Talton et al. 2011), villages (Emilien et al. 2012), and complex buildings and landscapes (Merrell and Manocha 2008). Grammatical generation is also used with natural languages to procedurally generate sentences (Kempen and Hoenkamp 1987) and dialogs (Ryan, Mateas, and Wardrip-Fruin 2016).

These works typically use specific grammars designed for the corresponding problems. A more generalized study on grammatical generation was carried out with the cellular programming model (McCormack 2005). Our work is inspired by the need from these and other works for a convenient and generalized tool for encoding grammatical PCGs, and therefore can potentially ease their implementation.

DSL for Games

Most commercial video games are built with game engines, which are not typically considered as DSLs themselves, but often contain scripting components that use programming languages, such as C++ for Unreal Engine (Epic Games (1998)) and C# for Unity Engine (Unity Technologies (2004)). DSLs extending from or interfacing with these languages are readily incorporated into game production.

The academic study of DSLs in game related context is common. Cutler et al. proposed a DSL for authoring solid model (2002). *Script Cards* is a visual programming language for enabling young people to create games and simulations (Howland, Good, and Robertson 2006). Tang et al. proposed a domain specific modeling language for serious games design modeling (2008). *Eberos GML2D* (Hernandez and Ortega 2010) was developed as a graphic DSL for modeling 2D video games using the Unreal Engine. Marques et al. built a DSL for RPG games that supports model-driven development (2012). *Video Game Description Language* (Schaul 2013) is a DSL for 2D tile-based games. Hastjarjanto et al. developed a declarative DSL embedded in Haskell for describing AI in video games (2013). *HyPED* (Osborn, Lambrigger, and Mateas 2017) studies modeling action games using a formal language. *Marahel* (Khalifa and Togelius 2017) was introduced as a DSL that directly supports common PCG tasks needed in 2D tile-based games.

Our work shares a close relationship with all the above DSL study, though with the unique goal of providing a DSL for generalized grammar-based PCG tasks. Two very closely related projects that share our same goal are *Tracery* (Compton, Filstrup, and Mateas 2014), a language to support grammar-based procedural story authoring, and *Ceptre* (Martens 2015), a language for encoding rules for interactive narratives and strategy games. Our work is different from both primarily in that GIGL supports direct user control of the stochastic components of grammars. In addition, GIGL code can be incorporated into existing C++ code at compile time, generating in-memory objects that have direct interface with a general purpose language.

Item Grammars and Item Trees

We refer to our proposed formalization of grammatical content generation as *item grammars*. Item grammars encode relationship between aspects of an object (e.g., parts of an item) through a set of rules that relate *nonterminals* (which represent high-level parts), to other nonterminals and *terminals* (which represent concrete details). The content generation process starts from a nonterminal and recursively expands it into a set of nonterminals or terminals by stochastically selecting applicable rules, until all branches reach terminals. This model is similar to probabilistic context-free grammars (PCFGs), but with the additional power of supporting context sensitive probabilities.

$$Monster := weakMonster()(P = 0.8)$$
$$\quad | \; strongMonster(Weapon)(P = 0.2)$$
$$Weapon := clubWeapon()(P = 0.7)$$
$$\quad | \; flailWeapon()(P = 0.3)$$

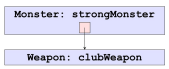


Figure 2: A simple item grammar for generating dungeon monsters (left) and the item tree for an example monster generated by the grammar (right).

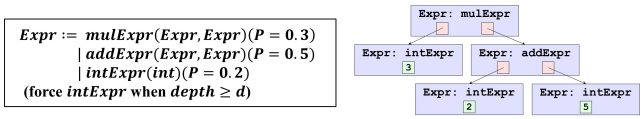


Figure 3: An item grammar for generating arithmetic quizzes (left) and the item tree for an example quiz generated by the grammar when $d = 2$ (right), which corresponds to $3 \times (2 + 5)$.

An item grammar encodes possible hierarchical structures within items. In addition, we use the term *item tree* to refer to the structure in an instance of generated item, which is analogous to a syntax tree for a CFG. Figure 2 shows an example item grammar for monsters in dungeon games (a simpler variants of Fig. 1) and one possible item tree. This grammar encodes monsters who may or may not have a weapon, which may be a club or flail weapon. Which type of monster is generated is determined by the expansion of a *Monster* (a nonterminal type) node, which can choose either *weakMonster* or *strongMonster* (two rules). The probability for each type being chosen is also provided in the specification of the rules. Here, applying the rule *strongMonster* further requires the expansion of its child, a *Weapon* nonterminal node, which uses information on the expansion rules for the *Weapon* nonterminal type in a similar fashion.

GIGL allows rule probabilities to depend on the context, which is important for customized control in PCG. For example, the item grammar in Fig. 3 generates math quizzes of integer additions and multiplications for elementary school students. To prevent the *Expr* nonterminals from expanding indefinitely we limit the size of the generated quizzes by adjusting the rule probabilities based on the current depth d (e.g. the probability to choose *intExpr* is 1.0 when the depth reaches d and 0.2 otherwise).

GIGL Framework

We propose *Grammatical Item Generation Language* (GIGL) as a DSL to encode item grammars and thus grammatical PCG. GIGL extends from C++ syntax, has direct interface with C++ and is implemented with C++ as an intermediate language. A detailed documentation of GIGL including a formal definition of its syntax can be found at `https://z.umn.edu/gigl`.

GIGL encodes grammatical PCG through two main aspects: an *item type* definition, which defines a data type for the (grammatical) item along with its generator, and the *generator configuration*, which allows customization of item generators. We show the general framework of GIGL through the *Monster* example, based on the item grammar described in the previous section (Fig. 2). A complete implementation of the Monster example (in GIGL), and its execution, is shown in Fig. 4.

Item Type Definition

An *item type* encodes a parametrized item grammar along with relevant item generator details (Line 3 - 23 in Fig. 4). Different types of blocks within the item type definition take different roles in this task. The *node* blocks declare the attributes on the nodes (e.g., a weapon's damage amount), *nonterminal* blocks declare the nonterminal types, and the *rule* blocks declare the rules and how attributes are computed when expanding through these rules (e.g., randomly setting the damage amount based on the weapon type).

Generator Configuration

The *generator configuration* allows the specification of the parameters of item grammars and the associated item generation processes. Line 27 - 29 and Line 34 - 36 show two different examples of generator configurations representing different monster difficulties via different rule probability settings. The code starting with "<*" and ending with "*>" is an expression that specifies a particular configuration, and "*generate ... with ...*" generates an item instance following the configuration, returning a C++ object of the specified item type. The GIGL code for the Monster example already shows the flexibility (easily tunable probabilities) and modularity (multiple instances of configuration without rewriting the type definition) provided by this system.

C++ Integration

GIGL's ability to integrate with C++ is designed on the code level, by having item generators generating C++ objects. We choose the approach of creating a DSL to integrate with an existing language instead of creating a library, because a language enables a more natural expression of design ideas in PCG, compared to passing in options, parameters, function pointers etc. as arguments to library functions. The top-right section of Fig. 4 shows C++ source code that directly interfaces with the GIGL code on the left. Two lists of monsters are generated with different configurations (corresponding to difficulties) and queried for their damage attributes, and then deleted. The generated monsters are used as C++ objects. The attributes can be queried (and potentially be modified) and the objects can be deleted with their pointers, in the same way as in C++ (highlighted in red).

GIGL Implementation

GIGL is implemented by using C++ as an intermediate language. We choose C++ as it provides a friendly and encapsulated interface and produces runtime efficient executables. GIGL is translated into C++ then compiled with C++ compilers. The GIGL-C++ translator is coded in *Silver* (Van Wyk et al. 2010), a domain specific programming language for creating new programming languages, and with *AbleC* (Kaminski et al. 2017), a Silver implementation of C for supporting the non-GIGL-specific part of the language (i.e. C/C++). The intermediate C++ layer ensures the code level integration of GIGL with C++.

Controlling Probabilistic Expansions

The Monster example demonstrates the ability to control the item generation process by configuring probabilities for various rule expansions. For more interesting and practical content generation, it is often necessary for designers to specify more advanced control on the probabilistic expansions.

```
1  giglstart;
2
3  gigltype DungeonMonster: {
4  node:
5    int damage;
6
7  nonterminal:
8    Monster;
9    Weapon;
10
11 rule:
12   Monster :=
13   | weakMonster:
14     { damage = GetRandInt(2, 5); }
15   | strongMonster: Weapon* weapon
16     { damage = GetRandInt(6, 9) + weapon->damage; }
17
18   Weapon :=
19   | clubWeapon:
20     { damage = GetRandInt(1, 3); }
21   | flailWeapon:
22     { damage = GetRandInt(11, 19); }
23 };
24
25 void GenerateEasyRoomMonsters(DungeonMonster** list) {
26   for(int i = 0; i < 10; i++)
27     list[i] = generate DungeonMonster with <* DungeonMonster:
28       Monster := weakMonster @ {0.8} | strongMonster @ {0.2},
29       Weapon := clubWeapon @ {1.0} *>;
30 }
31
32 void GenerateHardRoomMonsters(DungeonMonster** list) {
33   for(int i = 0; i < 10; i++)
34     list[i] = generate DungeonMonster with <* DungeonMonster:
35       Monster := weakMonster @ {0.6} | strongMonster @ {0.4},
36       Weapon := clubWeapon @ {0.7} | flailWeapon @ {0.3} *>;
37 }
```

```
int seed;
DungeonMonster* monster_set[10];

cout << "input seed: ";
cin >> seed;
RandInit(seed);
cout << "Monster stats in Easy Room #"
  << seed << endl;
GenerateEasyRoomMonsters(monster_set);
for (int j = 0; j < 10; j++) {
  cout << monster_set[j]->damage << "  ";
  delete monster_set[j];
}
cout << endl << endl;

cout << "input seed: ";
cin >> seed;
RandInit(seed);
cout << "Monster stats in Hard Room #"
  << seed << endl;
GenerateHardRoomMonsters(monster_set);
for (int j = 0; j < 10; j++) {
  cout << monster_set[j]->damage << "  ";
  delete monster_set[j];
}
cout << endl << endl;
```

```
input seed: 2
Monster stats in Easy Room #2
12  3  5  4  5  9  11  3  3  4

input seed: 2
Monster stats in Hard Room #2
28  3  5  4  24  2  11  3  3  8
```

Figure 4: The usage of GIGL in the Monster example. Left: the GIGL code (non-C++ reserved words marked in blue). Top-right: the C++ code that interfaces with GIGL (interactions marked in red). Bottom-right: an example input-output sequence.

GIGL allows those fine-grain levels of control, e.g., by allowing rule probabilities to be set through lambda expressions. We showcase this through an L-system tree generator problem (called the *Tree* example).

The L-system Tree Generator

L-systems are fractal systems constructed by grammars and are well-suited in generating plants (Lindenmayer 1968). Here we use an L-system with the grammar $F \rightarrow F[-F][+F]F$ (Fig. 6a - 6c) to generate trees. The grammar describes a sequential drawing motion, where $-$ means turning right, $+$ means turning left; brackets indicate preserving the last starting position. We translate the L-system grammar above into the following item grammar (concrete probability values omitted as GIGL parametrizes rule probabilities):

$$
\begin{aligned}
&\mathit{TreePart} := \\
&\quad |\mathit{ntTree}(\mathit{TreePart}, \mathit{TreePart}, \mathit{TreePart}, \mathit{TreePart}) \\
&\quad |\mathit{termTree}(\mathit{TreeSegment}) \ \ \text{(forced when } \mathit{depth} \geq n\text{)}
\end{aligned}
$$

where the nonterminal type *TreePart* corresponds to the F in the L-system grammar and we added a second rule *termTree*, so that the expansion may terminate into concrete tree segments (the terminal type *TreeSegment*) at a certain depth.

Examples from the GIGL implementation for the Tree example is shown in Fig. 5. The *Draw* on Line 5 is a function that operates recursively over the grammatical structure for rendering the tree. To determine the depth of a node, we ex-

```
1  ...
2  gigltype LTree{...}: {
3  ...
4  node:
5    void Draw();
6  ...
7  nonterminal:
8    TreePart(int depth, ...);
9
10 rule:
11   TreePart :=
12   | ntTree{double branch_deg}:
13     TreePart* bottom, TreePart* right,
14     TreePart* left, TreePart* top
15     {
16       generator {
17         bottom = generate TreePart(depth + 1, ...);
18         right = generate TreePart(depth + 1, ...);
19         left = generate TreePart(depth + 1, ...);
20         top = generate TreePart(depth + 1, ...);
21         ...
22       }
23       Draw {...}
24     }
25   | termTree: TreeSegment* seg { ... }
26 };
27
28 LTree* GenerateLTree(...) {
29   return generate LTree with <* LTree{...}:
30     TreePart := ntTree{45.0} @ {depth < n} | termTree *>;
31 }
```

Figure 5: Key fragments of GIGL code for the Tree example, of the version with deterministic configuration. Grey "..."s are omitted code. Full code for this and other examples is available at `https://z.umn.edu/gigl`.

(a) $n = 1$ (8x) (b) $n = 2$ (4x) (c) $n = 5$ (d) Stochastic (e) Tuned

Figure 6: (a)(b)(c) The results of the Tree example when applying the L-system rule deterministically up to some depth n, shown with both plain rendering and rich rendering. (d) Three trees generated with a less constricted randomization. (e) Three trees generated with a better tuned randomization. Both (d) and (e) have a depth bound of 5.

plicitly call the node generator with the `generate` command and increment the depth (lines 17 - 20). The depth information is needed for the depth-limiting behavior, which is further discussed in the following subsection.

Parametrization with Configure Parameters

The rule probabilities in item generation can be controlled in GIGL via arbitrary lambda expressions. For example, termination may be more likely at certain levels of depths in the tree, or even forced entirely. These expressions can depend on local information (e.g. the depth), in which case they must be evaluated when generating each node in the item tree (as opposed to when setting the configuration). In the Tree example, the `depth<n` on Line 30 is such a lambda expression, meaning the probability to further expand (select the *ntTree* rule) is 0 when the depth reaches n and 1 otherwise. This sets the termination depth to n.

The item generation process can be further parametrized by specifying parameters that can be set in the generator configuration (called *configure parameters*). Rule probabilities are by default configure parameters, additional ones can be declared in braces such as the `branch_deg` on Line 12, which means the turning angle for branches. It is set with a constant value 45 (degrees) on Line 30 here, but can in general be an arbitrary lambda expression. Fig. 6a - 6c show the results of this configuration, which are trees expanded to exactly the specified depths with the L-system grammar.

Stochastic Generation. The above implementation faithfully executes the L-system, but the results are deterministic and too symmetric. We can reduce the symmetry by occasionally skipping left or right branches. GIGL has direct support for this "maybe" type expansion. We can modify the generator calls for the `left` and `right` children of the *ntTree* rule (RHS of Line 18 and 19) to

```
generate<0.7> TreePart(depth+1, ...);
```

which specifies that there is 0.3 probability that the branch is empty. In addition, we modify the rule part (lambda expressions) of the generator configuration on Line 30 into:

```
TreePart :=
    ntTree{GetRandFloat(30.0,60.0)}
        @{depth<n?0.8:0.0}|termTree
```

to randomize branch angles, and allow early termination. The overall results can be seen in Fig. 6d.

Fine-tuning the Stochastic Expansion. The above results show clear variation, however, the variation is too drastic and not in a desired way. We prefer having less branches near the bottom than near the top, and thinner branches more likely to terminate early. We can achieve all these by simply passing in more sophisticated functions for the rule probabilities and the "maybe" probabilities. By replacing the probability for *ntTree* with `(depth<n?0.5+bottom_w/max_w:0.0)` and replacing the branch probability in the "maybe" generator calls with `(2-3*top_w/max_w)`, where *top_w*, *bottom_w* and *max_w* are the width at the top and bottom of this tree part and that at the bottom of the whole tree respectively, we can significantly improve the results as shown in Fig. 6e.

Enforcing Constraints

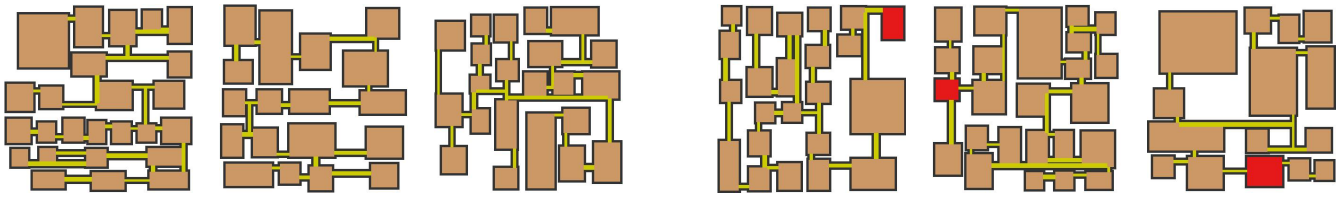
An important consideration when using grammars for PCG is the need to set hard constraints on rule expansions. In the Tree example, it was possible to constrain the depth by using lambda expressions to set rule probabilities. A more natural expression of these constraints would allow "pre-selecting" what rules are allowed based on local conditions. GIGL supports this paradigm though the use of *pre-selector* blocks with statements that directly constrain rule selections (e.g. forbid certain rules). We explore its use in a dungeon generator problem (called the *Dungeon* example).

The BSP Dungeon Generator

Following the approach in (Shaker et al. 2016), we use a grammatical approach to generate procedural dungeon levels based on binary space partitioning (BSP) trees, that recursively split space into two parts. In creating the dungeon, the each partition corresponds to a room (or set of rooms) and the division from each split is connected with a corridor. We encode this approach with the following item grammar:

$$
\begin{aligned}
&DungeonArea := \\
&\quad |hDivide(DungeonArea, DungeonArea, Corridor) \\
&\quad |vDivide(DungeonArea, DungeonArea, Corridor) \\
&\quad |tArea(Room) \\
&\quad (\text{forbid } vDivide \text{ when } w < 2s_{min}, \\
&\quad \text{forbid } hDivide \text{ when } h < 2s_{min}, \\
&\quad \text{forbid } tArea \text{ when } w > s_{max} \text{ or } h > s_{max}),
\end{aligned}
$$

where w and h denote the width and height of the current area of dungeon to generate. The values s_{min} and s_{max} control the size of the dungeons rooms by setting constraints



(a) Regular Dungeons (b) Boss Dungeons

Figure 7: Examples of dungeon generation (a) with no boss room and (b) with exactly one boss room. Brown indicates rooms, yellow indicates corridors, red indicates boss rooms.

on the minimum and maximum side-length of the single-room regions the splits are allowed to create. Example output from this grammar (implemented in GIGL and rendered with OpenGL in C++) is shown in Fig. 7a.

For generating interesting dungeons, we might want some dungeon rooms to be special, such as those containing the final boss of the level (referred to as *boss rooms*). We can make each dungeon to contain exactly one, randomized boss room also with the constraint mechanism. First, we add a boolean parameter *b* to indicate whether or not to generate a boss in this region. Second, we modify the implementation of *hDivide* and *vDivide* such that they pass *true* to one child area and *false* to the other as the value of *b* if the *b* of current node is true, and pass in both *false* values otherwise. Third, we add the following rule to the rule set for *DungeonArea*,

$|bossArea(BossRoom)$

(forbid $bossArea$ when $w > s_{max}$ or $h > s_{max}$ or $b = false$),

so that the expansion terminates into a boss room instead of a normal one when b is true. The result of this modification is shown in Fig. 7b.

Encoding Constraints with Pre-selectors

The GIGL implementation of the constraints in the Dungeon example can be done through the use of *forbid* statements in the pre-selector block as follows:

```
if (w < 2 * s_min) forbid vDivide;
if (h < 2 * s_min) forbid hDivide;
if (w > s_max || h > s_max)
  forbid tArea, bossArea;
else if (b)
  forbid<transferto bossArea> tArea;
else
  forbid<transferto tArea> bossArea;
```

Here `forbid` prevents a rule from being selected by setting its probability to 0 and, by default, re-normalizing the remaining probabilities. The option `transferto X` instead specifies that the forbidden probability is added to the probability of some other rule `X`.

Analyses and Discussions

To facilitate a quantitative evaluation of GIGL, we created the C++ counterparts for the Monster, Tree, and Dungeon examples. C++ is chosen for comparison because GIGL has C++ interfaces, which makes it a direct alternative to C++. For the C++ versions, we adopt implementations that are natural to each PCG problem (e.g. representing the two possible monster configurations with a boolean). This grants a slight advantage to C++ in comparing code sizes and run times, but at the cost of less generalizability.

	Original			Compressed		
	C++	GIGL	Ratio	C++	GIGL	Ratio
Monster	1.65	0.88	0.533	0.66	0.49	0.735
Tree	3.89	2.33	0.599	1.25	0.86	0.693
Dungeon	11.6	9.9	0.853	2.17	1.97	0.908

Table 1: The file size comparison (sizes are in KB) between C++ and GIGL source files and their compressed files.

Code Size Analyses

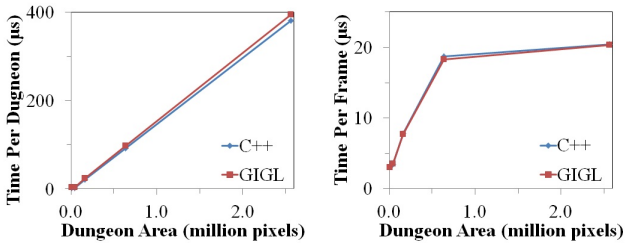
One metric for measuring a language's expressiveness is the size of the code needed to encode the algorithm. Here we compare the size of the raw C++ and GIGL implementations. We also compare the corresponding compressed file size (by *7-Zip* to default zip format), to mitigate the dependency on simple textual aspects such as keyword lengths.

The results of code size tests are displayed in Table 1. The GIGL code is shorter, especially for the Monster and Tree example, where GIGL saves 40% to 50% of the code. Even for the compressed version, GIGL still has clear advantages. The smaller saving ratio for the Dungeon example is primarily due to large non-grammar parts (such as positioning corridors, rendering etc.) being unaffected by GIGL.

Runtime Analyses

We run a performance analysis using the Dungeon example (with a boss room), by comparing the run times of corresponding C++ and GIGL code. We vary the side-length of the dungeon between 100 and 1600 pixels. All runtime tests are performed on a machine with an Intel Core 4.0 GHz processor. In all cases, the GIGL run time was very short, with even complex objects with hundreds of components (rooms and corridors) being generated under a millisecond and their access times being only on the order of microseconds.

Generate Time. We measure the time taken to generate the dungeon by disabling all rendering related operations and repeatedly generating and deleting random dungeons. The results are shown in Fig. 8a. The GIGL code is nearly as computationally efficient as C++, with both showing a linear relation between runtime and dungeon area, and with little performance difference between the two approaches.



(a) Generate Time (b) Access Time

Figure 8: **Runtime Analyses** A comparison of (a) time for generating each dungeon and (b) time accessing a dungeon for rendering purposes. Each chart shows the run times using GIGL and C++ for increasing dungeon complexity which scales with dungeon area. Run times are tested for 1 million generations or accesses, and are averaged across 8 runs.

	C++			GIGL		
	Size	Gen.	Acc.	Size	Gen.	Acc.
ViewCull	11.6	381.3	20.4	9.9	394.2	20.3
Hierarchy	10.9	375.9	81.9	9.3	391.1	81.9
Flat	9.4	1392.0	75.8	-	-	-

Table 2: The file sizes (in KB), and run times (in µs) for generating (Gen.) and accessing (Acc.) on 1600×1600 (pixels) dungeons, for different approaches to the Dungeon example, and in C++ and GIGL. GIGL automatically creates hierarchies therefore is not tested for the *Flat* approach.

Access Time. We separately measure the performance of accessing elements of the procedural generated items after they have been created. Here, we create a single dungeon, and write the central 600×600 of it to the frame buffer. Here the frame buffer is not swapped to screen, to avoid adding a same large overhead to all compared tests. As shown in Fig. 8b, the results on this set of tests show almost no difference between GIGL and C++, which indicates that the implementation of item structures in GIGL is efficient for item access. Note that we are able to achieve a sub-linear growth in access time by incorporating a spatial data structure in the dungeon's representation. The structures captured in GIGL item types naturally support these kinds of hierarchical acceleration data structure as discussed below.

Case Study: Hierarchical Accelerations

A key feature of GIGL is that the items generated automatically retain the underlying hierarchical structures represented by the grammar. In many cases this structure can be exploited for optimization. Here, we can exploit the tree-like structure of the dungeon both to accelerate item generation for finding corridor connections, and to accelerate item accessing for view-culling. Table 2 compares C++ and GIGL implementations of these accelerations where *Hierarchy* accelerates only the generation, *ViewCull* accelerates both, and *Flat* accelerates none. The runtime results show clear benefits of exploiting hierarchical structures both in generation and accessing. This demonstrates the importance of retaining those structures (i.e. item trees) in the generated objects. In addition, in all tests, GIGL is smaller in code size and runs almost as fast as its C++ counterparts. The C++ code can be made as small as GIGL only by removing support for hierarchal structures (*Flat*), resulting in code that is much slower. This further supports the conclusion that GIGL is effective in efficiently expressing grammatical PCG while maintaining good runtime performance.

Conclusion

In this paper, we propose GIGL, a DSL with expressive encoding for grammatical PCG that has direct interface with C++, where generators generate C++ objects maintaining the grammatical structures. GIGL provides important features that allow users to control and constrain the stochastic content generation process, e.g. lambda expressions in setting configure parameters, and forbid statements in rule pre-selectors. In addition, GIGL is implemented to be compilable, and GIGL code is shown to have high runtime performances (close to C++) with relatively small code size.

Our work has limitations which we wish to address in future work including adding robust type-checking on the GIGL source code, establishing compatibilities with other domain specific language extensions of C/C++, and adding more domain specific features such as support for machine learning and automatic constraint solvers. As evaluating the usability of a DSL is an open problem in the field, our future work also includes exploring options of user studies on effectiveness of GIGL in authoring.

Acknowledgment

This work has been supported in part by the NSF through grants #CHS-1526693 and #CNS-1544887. In addition, the discussions and technical supports from Travis Carlson and Professor Eric Van Wyk in the MELT group are appreciated.

References

Arnold, J., and Alexander, R. 2013. Testing autonomous robot control software using procedural content generation. In *International Conference on Computer Safety, Reliability, and Security*, 33–44. Springer.

Claessen, K., and Hughes, J. 2011. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices* 46(4):53–64.

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

Cutler, B.; Dorsey, J.; McMillan, L.; Müller, M.; and Jagnow, R. 2002. A procedural approach to authoring solid models. In *ACM Transactions on Graphics (TOG)*, volume 21, 302–311. ACM.

Emilien, A.; Bernhardt, A.; Peytavie, A.; Cani, M.-P.; and Galin, E. 2012. Procedural generation of villages on arbitrary terrains. *The Visual Computer* 28(6-8):809–818.

Epic Games. 1998. Unreal Engine. `https://www.unrealengine.com/en-US/what-is-unreal-engine-4`. Accessed: 2017-09-11.

Germer, T., and Schwarz, M. 2009. Procedural arrangement of furniture for real-time walkthroughs. In *Computer Graphics Forum*, volume 28, 2068–2078. Wiley Online Library.

Hastjarjanto, T.; Jeuring, J.; and Leather, S. 2013. A dsl for describing the artificial intelligence in real-time video games. In *Proceedings of the 3rd International Workshop on Games and Software Engineering: Engineering Computer Games to Enable Positive, Progressive Change*, 8–14. IEEE Press.

Hernandez, F. E., and Ortega, F. R. 2010. Eberos gml2d: a graphical domain-specific language for modeling 2d video games. In *Proceedings of the 10th Workshop on Domain-Specific Modeling*, 1–1. ACM.

Howland, K.; Good, J.; and Robertson, J. 2006. Script cards: a visual programming language for games authoring by young people. In *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*, 181–186. IEEE.

Kaminski, T.; Kramer, L.; Carlson, T.; and Van Wyk, E. 2017. ablec: Extensible specification of c using the silver attribute grammar system.

Kempen, G., and Hoenkamp, E. 1987. An incremental procedural grammar for sentence formulation. *Cognitive science* 11(2):201–258.

Khalifa, A., and Togelius, J. 2017. Marahel: A language for constructive level generation.

Krecklau, L., and Kobbelt, L. 2012. Interactive modeling by procedural high-level primitives. *Computers & Graphics* 36(5):376–386.

Lindenmayer, A. 1968. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of theoretical biology* 18(3):280–299.

Marques, E.; Balegas, V.; Barroca, B. F.; Barisic, A.; and Amaral, V. 2012. The rpg dsl: a case study of language engineering using mdd for generating rpg games for mobile phones. In *Proceedings of the 2012 workshop on Domain-specific modeling*, 13–18. ACM.

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

McCormack, J. 2005. A developmental model for generative media. In *European Conference on Artificial Life*, 88–97. Springer.

Merrell, P., and Manocha, D. 2008. Continuous model synthesis. In *ACM transactions on graphics (TOG)*, volume 27, 158. ACM.

Merrick, K. E.; Isaacs, A.; Barlow, M.; and Gu, N. 2013. A shape grammar approach to computational creativity and procedural content generation in massively multiplayer online role playing games. *Entertainment Computing* 4(2):115–130.

Osborn, J. C.; Lambrigger, B.; and Mateas, M. 2017. Hyped: Modeling and analyzing action games as hybrid systems. In *Thirteenth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Parish, Y. I., and Müller, P. 2001. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 301–308. ACM.

Ryan, J.; Mateas, M.; and Wardrip-Fruin, N. 2016. Characters who speak their minds: Dialogue generation in talk of the town. *Proc. AIIDE*.

Schaul, T. 2013. A video game description language for model-based or interactive learning. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, 1–8. IEEE.

Shaker, N.; Nicolau, M.; Yannakakis, G. N.; Togelius, J.; and O'neill, M. 2012. Evolving levels for super mario bros using grammatical evolution. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, 304–311. IEEE.

Shaker, N.; Liapis, A.; Togelius, J.; Lopes, R.; and Bidarra, R. 2016. Constructive generation methods for dungeons and levels. In *Procedural Content Generation in Games*. Springer. 31–55.

Shaker, N.; Togelius, J.; and Nelson, M. J. 2016. *Procedural Content Generation in Games*. Springer.

Smelik, R. M.; Tutenel, T.; de Kraker, K. J.; and Bidarra, R. 2011. A declarative approach to procedural modeling of virtual worlds. *Computers & Graphics* 35(2):352–363.

Smith, G.; Treanor, M.; Whitehead, J.; and Mateas, M. 2009. Rhythm-based level generation for 2d platformers. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, 175–182. ACM.

Smith, A. R. 1984. Plants, fractals, and formal languages. *ACM SIGGRAPH Computer Graphics* 18(3):1–10.

Talton, J. O.; Lou, Y.; Lesser, S.; Duke, J.; Měch, R.; and Koltun, V. 2011. Metropolis procedural modeling. *ACM Transactions on Graphics (TOG)* 30(2):11.

Tang, S.; Hanneghan, M.; Hughes, T.; Dennett, C.; Cooper, S.; and Sabri, M. A. 2008. Towards a domain specific modelling language for serious game design. In *6th International Game Design and Technology Workshop, Liverpool, UK*.

Togelius, J.; Kastbjerg, E.; Schedl, D.; and Yannakakis, G. N. 2011. What is procedural content generation?: Mario on the borderline. In *Proceedings of the 2nd international workshop on procedural content generation in games*, 3. ACM.

Tomás, A. P., and Leal, J. P. 2013. Automatic generation and delivery of multiple-choice math quizzes. In *International Conference on Principles and Practice of Constraint Programming*, 848–863. Springer.

Toto, F. S. G., and Vessio, G. 2014. A probabilistic grammar for procedural content generation. In *Sixth Workshop on Non-Classical Models of Automata and Applications (NCMA 2014)*, 31.

Unity Technologies. 2004. Unity. `https://unity3d.com/`. Accessed: 2017-09-11.

Van Wyk, E.; Bodin, D.; Gao, J.; and Krishnan, L. 2010. Silver: An extensible attribute grammar system. *Science of Computer Programming* 75(1-2):39–54.