Domain-agnostic Procedural Content Generation Can Be Done Declaratively

Denise Angilica[1], Stefano Germano[2] and Giovambattista Ianni[1]

[1]University of Calabria
[2]University of Oxford

Abstract

Procedural content generation is applied in the development process of many commercial games: automatically generated contents are delivered to players in order to offer a constantly changing user experience and enrich the game itself. Designing algorithms for content generators can be a tedious job. The product of this work is often too domain specific and offers limited reusability and customizability. Declarative approaches to content generation, more properly defined as Declarative Content Specification (DCS) techniques, like the ones based on Answer Set Programming (ASP), promise to overcome some of these drawbacks, since they allow focusing on describing content requirements rather than programming ad-hoc generation engines. Also, DCS speed up the prototype generation techniques themselves. In turn, DCS techniques struggle to gain momentum mainly because of lack of integration with game engines. Furthermore, the promise of generality and reusability is neutralized by the burden of wiring and adapting declarative content specifications to the context of the game at hand. In this work, we report about our progress toward a general DCS module working in the Unity game engine, and integrated in an existing asset for declaratively defining AI modules. We illustrate both the design and runtime workflow of this module, and how game content developers could use it for devising their own content generation schemes. For this purpose, an example highlighting the advantages of this approach is described.

Keywords

Answer Set Programming, Procedural Content Generation, Game Content Generation, Artificial Intelligence in Games, Computational Intelligence in Games, Declarative Content Specification

1. Introduction

Procedural Content Generation (PCG) [1] is an important tool for modern video game development, and is commonly used in both triple-A (i.e., high-budget games) and indie games. An effective PCG framework enables the development of fresh game content without explicitly handcrafting it. Instead, a software algorithm is executed, generating output that is integrated into the game as landscapes, playable stages, open worlds, and other elements commonly referred to as game artifacts. The game design workflow usually involves a number of professional profiles, whose jobs can partially overlap. In particular, the game *designer* is in charge of manually designing and combining game artifacts, while the *programmer* writes general

CILC'23: 38th Italian Conference on Computational Logic, June 21–23, 2023, Udine, Italy
denise.angilica@unical.it (D. Angilica); stefano.germano@cs.ox.ac.uk (S. Germano); giovambattista.ianni@unical.it (G. Ianni)
0000-0002-4069-978X (D. Angilica); 0000-0001-6993-0618 (S. Germano); 0000-0003-0534-6425 (G. Ianni)

purpose code, Artificial Intelligence (AI) code, and, in our context, content generation code.

Designers and programmers have usually a strict collaboration, and interact in order to produce the right content. There can be cases in which the designer has little inspiration and cannot converge to a concrete game world or to suitable new levels, nor explore novel ideas, etc. In this setting *PCG* might be very beneficial; however, finding a good content generation scheme might be a big burden on programmers' shoulders.

Content generation schemes are quite often *domain dependent* and thus reusability is an issue. In this respect, the adoption of machine learning generated content is quite appealing, especially the usage of generative adversarial networks (GANs) [2]. It is however known that GANs are not at ease when it is necessary to respect geometric, 3D perspectives, and counting constraints [3]. Also, one has to consider that training times have impact on the overall game design time, and that little modifications on the requirements necessitate further re-training. In other development contexts, designers extensively collaborate with a programmer in order to modify the content creation algorithm, so that the generated game artifacts fit to the original ideas and description. In both settings above, input ideas about the game world come from designers in terms of high-level rules and constraints, and such information is then used by programmers to encode algorithms which should generate the content specified; this, however, means that the programmer must devise not only what is supposed to be generated, but also the procedural algorithms in charge of the generation task.

It turns then out that a general technique, which is reusable and decoupled from the specific game domain and visual appearance, and results to be accessible to non-programmers and suitable for rapidly prototyping game content, can be significantly of help. In this respect, logic-based declarative tools, such as Answer Set Programming, can be a game changer, as they limit, if not eliminate, the need for imperative code, thus achieving the above benefits in several aspects. On the one hand, a skilled game designer can declaratively express quantitative and qualitative desiderata in terms of ASP specifications; on the other hand, programmers themselves can define content generation strategies without the burden of programming detailed algorithms via imperative languages.

This motivated the usage of ASP as the declarative core for expressing preferences used for procedural content generation in [4, 5] and [6]. Other examples are expressing requirements for auto-generated architectural buildings [7] or for the placement thereof [8], and narrative generation [9, 10]. In a sense, ASP can be used for evolving traditional PCG techniques to the notion of what might better be defined as Declarative Content Specification (DCS) [4]: this new notion stresses the fact that declarative specifications can be easily modified and incremented with new knowledge at will.

The above contributions prove the potential of ASP in content generation, yet they may be seen as specific for a given type of game or for a specific type of content. In this paper we overview a domain agnostic approach at DCS aiming to *a)* gain generality, *b)* offer a better integration with commercial game engines, *c)* exploit the possibility of using new performant incremental ASP solvers [11]. The proposed DCS Module is expected to work within our already available ThinkEngine [12], a tool for building and deploying AI modules within the Unity game engine.

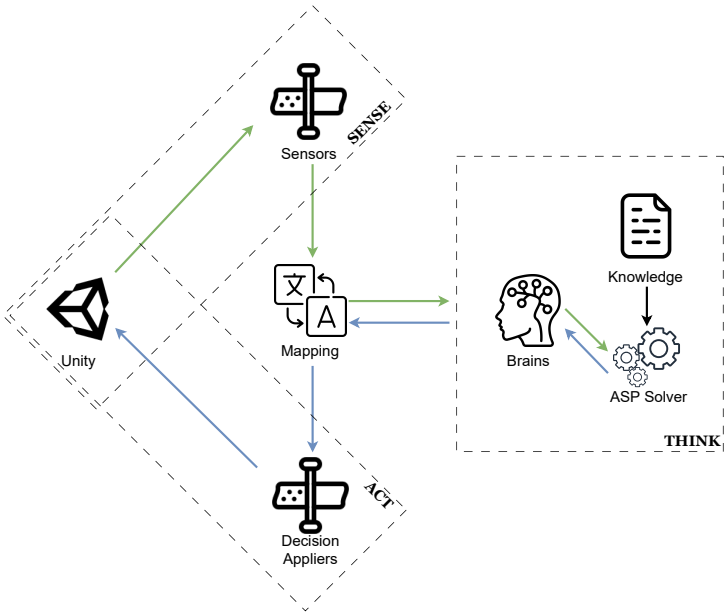

Figure 1: The DCS Module run-time architecture.

2. ThinkEngine: an Overview

Figure 1, adapted from the description of our ThinkEngine tool [12] shows our proposal of an integration scheme between a reasoner based on declarative specifications and a game engine. A number of so-called *brains* are able to interact with a video game scene.

An AI developer can use the ThinkEngine by first identifying objects and/or parts of the game logic that are to be connected to brains; *sensor readers* are wired from the game scene to brains, and *actuators* from brains to the game scene. A brain consists of a simple high-level specification that describes decision criteria: one can add rules, constraints, soft constraints and other declarative statement types. A brain reasoning task can be then triggered on specific events or on a cyclic basis. *Reactive brains* produce immediate modifications on the game, while *Planning brains* can be used to synthesize complex execution plans or, in principles, even arbitrarily reprogram reactive behaviors of artificial players.

In the case of our ThinkEngine tool, reasoning modules are implemented using declarative specifications written using ASP; an ASP solver executes such specifications, then its outputs are converted into actions or plans to be executed on the game world. ThinkEngine is implemented respecting a paradigm-agnostic stance: a wiring infrastructure for connecting other solvers based on other declarative paradigms (PDDL, SMT or others) is provided. In the specific setting of ThinkEngine, most of the interoperability burden is implemented within the EmbASP library [13]. The ThinkEngine works as a plugin for Unity [14], the known cross-platform game engine primarily used to develop video games or simulations for more than 20 different platforms like mobile, computers, and consoles. From the software development perspective, the presence of brains based on declarative tools implies several potential benefits, like:

- declarative tools can be used for defining, in a very short time, different aspects of game AI, like: modelling of non-player characters (NPC) with spatio-temporal reasoning based AIs, automated game resource management, path planning, automated narrative generation based on qualitative descriptions, dynamic dialogue systems, and in particular, one can devise DCS policies;
- non programmers can participate in the implementation of decision-making modules;
- the strong decoupling between declarative decision-making and the game runtime makes much easier to separate and parallelize the development of both;
- the decoupling from the game engine itself allows standalone testing, isolated debugging and optimization.

In general, the above advantages pair up with known barriers to the adoption of declarative tools in demanding applications and in the video game application settings in particular. Such barriers are mostly related to the actual declarativity of the formalism at hand, its evaluation efficiency, and the integration methodology with applications. ThinkEngine constitutes an effort aiming to close such said gaps.

3. Design-time Workflow of DCS Module

In the same spirit of other features of the ThinkEngine, with the DCS Module we aim to equip game designers with a content specification tool that emphasizes benefits of declarative formalisms while smoothing some of their drawbacks. Let us assume, for instance, that a game designer intends to automatically generate a game level for a platform game starting from a high-level specification. At design-time creators are provided with graphical and object-oriented tools that can be used to identify which game artifacts (*Prefab* in the Unity terminology) should be used while generating the game-world. Prefabs need to be detailed in their characteristics in order to be well-fitted in the generated level: our DCS Module offers general purpose tools to describe every feature of a game artifact, including their relation with other game elements, in a way that requires the least possible effort from the game designer. Alongside, a domain-agnostic knowledge base *KB* is provided. *KB* consists of an ASP encoding meant to work with a variety of games whose world can be generated, somehow, strip by strip (i.e., they are based on linear levels [15]). When needed, *KB* can be adjusted based on peculiarities of the game: one can add or delete rules or simply modify existing ones. The outcome of the DCS process can be fine-tuned by expressing some desiderata on the aspect of the generated world. DCS Module will enrich the ThinkEngine with a new type of brain that is the *Content Brain*. At design-time a Content Brain *a)* collects the prefabs to be used for the level generation; *b)* provides access to the default knowledge base; *c)* offers debug features.

4. Run-time Workflow of DCS Module

Once creators are satisfied with their configuration of Content Brains, these are used at game run-time following three steps, as follows. In the case of a platform game, a Content Brain will

repeatedly invoke an ASP solver, producing each time a new level strip, which is tiled next to the previous one.

Generation of additional background knowledge based on prefabs features. Based on configurations made at design-time, crucial information of prefabs, like, e.g., if the prefab is dangerous for the player character or which prefabs can be placed near each other, are translated in logical assertions *F*. For instance, one can disallow the presence of fire tiles above water tiles, etc.

Execution of the solver over the background knowledge. The knowledge base *KB*, whether it is the default or a custom one, is fed in input to the solver together with *F*. Thanks to the adoption of incremental solvers, the level generation is expected to be fast enough to be performed while the game is running. A random level, respecting the specification given, is created at each run.

Instantiation of prefabs depending on the answer set obtained. Once the solver derives a prefabs configuration for a single level strip, the ThinkEngine will automatically add to the game scene the selected artifacts without requiring any further development of glue code.

5. An Example

The DCS Module can be useful in several contexts, like general map generation for 2D/3D games. In the case of platform games, a foundational element is the possibility to express how the different available assets can be arranged in relation to each other. ASP offers a remarkably simple and flexible way to express such information. Assuming a platform level is made of consecutive, vertically arranged, strips, each of which made by a rectangular grid of tiles, compatibility between different assets can be expressed with facts like:

$$\text{compatible}(AssetA, Direction, AssetB).$$

to express that asset $AssetA$ can appear next to asset $AssetB$ w.r.t. direction $Direction$.

ASP rules can also be used to enhance the background *KB* and to streamline the modelling task of the designer. For instance, a simple rule can define the reverse compatibility:

$$\text{compatible}(AssetB, right, AssetA) \leftarrow \text{compatible}(AssetA, left, AssetB).$$

Defining the possible assignments of assets to tiles is straightforward. For instance, given the unary predicates *tile* and *asset*, a simple choice rule, like the following, defines the search space of possible assets that can fit a given tile:

$$1 \leq \{\text{contains\_asset}(Tile, Asset) : \text{asset}(Asset)\} \leq 1 \leftarrow \text{tile}(Tile).$$

where contains_asset$(Tile, Asset)$ expresses that tile $Tile$ contains asset $Asset$.

Besides the ease of defining possibly complex content policies and other abovementioned advantages, it is worth observing other interesting aspects. The content creators can easily test and debug different specifications and can even change them while running the game to check in real-time the effects of new rules. This streamlines the task of content creation and allows a more effective use of the designer and of the programmer time.

Additionally, specifications can be easily customized when special cases are needed. For instance, if a tile is required to have a given property, like, e.g., *fordability*, and specific assets

must be selected based on the presence of this property, negation-as-failure can be used to effortlessly define that. For instance, asset assignment can be defined as follows:

$$1 \leq \{\text{contains\_asset}(Tile, Asset) : \text{fordable\_asset}(Asset)\} \leq 1 \leftarrow \text{tile}(Tile), \text{property}(Tile, \text{fordable}).$$
$$1 \leq \{\text{contains\_asset}(Tile, Asset) : \text{nonfordable\_asset}(Asset)\} \leq 1 \leftarrow \text{tile}(Tile), \mathit{not}\ \text{property}(Tile, \text{fordable}).$$

6. Conclusions

Our DCS Module for ThinkEngine candidates as one of the first game/domain-agnostic DCS tool. Making ThinkEngine equipped with its new DCS Module available for the Unity game engine could make the PCG process easier for game designers. The tool appears to have good chances in being efficient enough to be used in a number of games whose levels can be randomly generated at run-time. We are currently working on *a)* refining the default knowledge base; *b)* identifying common prefabs' features in order to further reduce the manual burden for game designers; *c)* assessing the DCS Module actual performance both in terms of visual quality of the generated levels and of the time required to generate them.

Acknowledgments

This work was partially supported by: the PNRR MUR project PE0000013-FAIR, Spoke 9 - Green-aware AI – WP9.1; the project "Declarative Reasoning over Streams" (CUP H24I17000080001) – PRIN 2017; the LAIA laboratory (part of the SILA laboratory network at University of Calabria); eBay; Samsung Research UK; Siemens AG, and the EPSRC projects ConCur (EP/V050869/1) and UK FIRES (EP/S019111/1).

References

[1] N. Shaker, J. Togelius, M. J. Nelson, Procedural Content Generation in Games, Computational Synthesis and Creative Systems, Springer, 2016.
[2] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, Y. Bengio, Generative adversarial nets, in: NIPS, 2014, pp. 2672–2680.
[3] A. Osokin, A. Chessel, R. E. Carazo-Salas, F. Vaggi, GANs for biological image synthesis, in: IEEE ICCV, 2017, pp. 2252–2261.
[4] F. Calimeri, S. Germano, G. Ianni, F. Pacenza, A. Pezzimenti, A. Tucci, Answer set programming for declarative content specification: A scalable partitioning-based approach, in: AI*IA, 2018.
[5] X. Neufeld, S. Mostaghim, D. Perez-Liebana, Procedural level generation with answer set programming for general video game playing, in: CEEC, 2015, pp. 207–212.
[6] A. M. Smith, M. Mateas, Answer set programming for procedural content generation: A design space approach, IEEE Trans. Comput. Intell. AI Games 3 (2011) 187–200.

[7] L. van Aanholt, R. Bidarra, Declarative procedural generation of architecture with semantic architectural profiles, in: CoG, IEEE, 2020, pp. 351–358.
[8] M. Certický, Implementing a wall-in building placement in starcraft with declarative programming, CoRR abs/1306.4460 (2013).
[9] J. Robertson, R. M. Young, The general mediation engine, Experimental AI in Games: Papers from the 2014 AIIDE Workshop. AAAI Technical Report WS-14-16 10 (2014) 65–66.
[10] C. Dabral, C. Martens, Generating explorable narrative spaces with answer set programming, in: L. Lelis, D. Thue (Eds.), Proceedings of the Sixteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2020, virtual, October 19-23, 2020, AAAI Press, 2020, pp. 45–51. URL: https://ojs.aaai.org/index.php/AIIDE/article/view/7406.
[11] F. Calimeri, G. Ianni, F. Pacenza, S. Perri, J. Zangari, ASP-based multi-shot reasoning via DLV2 with incremental grounding, in: PPDP, ACM, 2022, pp. 2:1–2:9.
[12] D. Angilica, G. Ianni, F. Pacenza, Declarative AI design in Unity using answer set programming, in: CoG, IEEE, 2022, pp. 417–424.
[13] F. Calimeri, S. Germano, G. Ianni, F. Pacenza, S. Perri, J. Zangari, Integrating rule-based AI tools into mainstream game development, in: RuleML+RR, volume 11092, Springer, 2018, pp. 310–317.
[14] Unity 3D game engine, Last accessed: Jan 2023. URL: https://unity3d.com/unity.
[15] S. Dahlskog, J. Togelius, M. J. Nelson, Linear levels through n-grams, in: MindTrek, 2014, pp. 200–206.