STAIRS 2016
D. Pearce and H.S. Pinto (Eds.)

doi:10.3233/978-1-61499-682-8-209

Design Space Descriptions for Logical Generation of Content

Thomas SMITH [a,b], Julian PADGET [a] and Andrew VIDLER [b]
[a] *Centre for Digital Entertainment, University of Bath, UK*
[b] *Ninja Theory Ltd., Cambridge, UK*

Abstract. This paper describes our research-in-progress into providing content-agnostic procedural content generation for computer games via answer set programming and a formal semantic description of the design space. Existing generative approaches are often closely coupled to a particular form of content and/or game, limiting opportunities for re-use — a dungeon level generator in one game is substantially different to a sci-fi weapon generator in another. We propose an approach involving a re-usable generator system that is able to produce and refine a model of any structured content from formal desired properties produced by designers. As an intermediate step towards this system for multi-purpose content generation we present initial work on generating content within a commercial game engine, using answer set programming to generate varied and challenging combat progressions for third-person action/combat games. By using semantic models as input for a content-agnostic generation system, we hope to provide more domain-general content generation.

Keywords. procedural content generation, generative methods, answer set programming, video games

1. Introduction

Procedural content generation (PCG) is 'the algorithmic generation of game content with limited or no human contribution' [1], and is increasingly used by games companies to provide diverse and varied content. There is a wide range of generation approaches, and most procedural generators in current commercial games and research projects are highly bespoke: designed for generating a particular kind of content within a specific context. This lack of generality limits the scope for re-use of generator code or systems in other contexts, and bespoke implementations can make it complex to refine or alter the output of an existing generator without full understanding of its internal processes.

We propose a design for a 'content-agnostic' generation approach — i.e. a system whereby the specification of the domain of desired output is not implicitly encoded within the generator implementation, but provided as an input to a *general content generator*. Output from the system will be valid descriptions of instances of the specified content, which could vary from level layouts to branching narrative skeletons to complete descriptions of the gameplay environment. The development of a general content generator is described as an open research challenge in PCG in a recent overview of the field [1], and potentially a step towards the 'grand goals' of developing *Multi-level, Multi-content PCG* and *Generating Complete Games*.

Our proposed solution consists of two parts: a formal approach to defining the requirements that shape the design space of the desired content, and a system for converting these requirements into formal constraints and then generating content within the described space via a constraint satisfaction approach. An existing knowledge representation approach such as the Web Ontology Language (OWL [2]) may be appropriate for provision of the formal domain description, whilst previous PCG research [3] indicates that Answer Set Programming (ASP; a declarative logic programming formalism for solving problem encodings composed of facts and rules) may be suitable for selecting valid content descriptions from the translated constraints.

Section 2 provides a brief overview of the existing research in these areas, with the proposed approach more fully detailed in Section 3, giving description of both the end user workflow and the generator mechanisms. Our work to date focuses on the constraint satisfaction part of the system, and Section 4 presents our content generation for a commercial engine via integration of an ASP solver and handwritten problem encodings. Finally, in Section 5 we lay out our intentions for further development of the system.

We suggest that a general content generator developed in this manner would be able to provide output comparable to other existing generators, thereby reducing the need for novel bespoke implementations. This reusable approach could then benefit from improvement across multiple projects' development cycles, rather than the 'disposable' single-output approach that is currently common. End users of the generator would also benefit in that familiarity or expertise with the system could also be transferable into new contexts, and previous domain space definitions re-used.

2. Background and Related Work

This research brings together two distinct areas: techniques for formal knowledge representation, particularly for game development; and approaches for procedural content generation, specifically ASP (as detailed by Smith et al. [3]) and iterative refinement [4].

2.1. Semantics and Formal Models

For generating content within a specific domain, the system will require a complete description of the desired design space. To capture designers' intentions regarding the structure and interrelations within the output, this must go beyond surface-level position and type information and allow specification of the 'meaning' behind content and relationships (i.e. semantics). Kessig et al. [5] describe this application of semantic data as a way of improving consistency between appearance and affordance in virtual worlds.

A few domain specific languages (DSLs) for game production already exist, most notably Puzzlescript[1] and VGDL[2] (as in [6]). However, the domains these languages specify are narrow in scope, both restricted to small two-dimensional gameplay spaces. A more general semantic knowledge representation language such as OWL [2] allows for specification of ontologies relating to any domain by describing classes, entities, and their relationships within that domain. There are several pre-existing editors and reasoners within the OWL development ecosystem, and the ontology inheritance mechanism allows for extension from pre-defined 'upper ontologies' describing common concepts.

[1] http://puzzlescript.net/ – Puzzlescript, accessed 19 May 2016
[2] http://www.gvgai.net/vgdl.php – Video Game Description Language, accessed 19 May 2016

2.2. Answer Set Programming

Answer set programming (ASP) is a declarative logic programming paradigm. Problems are encoded in terms of facts and rules associated with the problem space, and after grounding the rules across each variable, a logic solver is used to select valid outputs from this constraint-bound space of possible solutions [7]. This potential for integrated verification is useful for the generation of content which must satisfy hard constraints, such as requiring that a valid solution for a puzzle exists or that all regions of a generated map are correctly connected [6]. Previous research into ASP for PCG proposes a 'design space' approach where the constraints on desired properties of the generated artefacts are specified directly within the ASP problem encoding [3].

Given a sufficiently complete set of restrictions on the output space, ASP can also assess edited or externally provided content to highlight constraint violations and suggest corrections, providing a mixed-initiative content production approach [8,9,10]. However, monolithic sufficiently complete descriptions of complex domains can be complex to author and have high computation costs, so it may be appropriate to split the encoding into multiple problems [11] or use an automated approach for ASP code authoring.

2.2.1. ASP with DSLs and OWL

As the network of constraints on a given problem becomes more complex, the problem becomes less suitable for manual authoring approaches, and so a number of simplified interfaces and translations from other languages have been developed.

Novel DSLs such as Inst*AL* [12] allow for reasoning about institutions by automatically translating a given Inst*AL* representation of an institutional modelling scenario to ASP. ASP can also be used to generate content for existing DSLs, as in the approach by Neufeld et al. to producing game levels from VGDL descriptions [6]. Finally, Gaggl et al. [13] present a mechanism for translating OWL reasoning problems into ASP via bounded model semantics, in order to take advantage of ASP's enumeration capabilities. Each of these approaches serves to simplify complex or repetitive ASP authoring tasks.

2.3. Iterative Refinement

Another approach to reducing the complexity of generation tasks is to decompose the problem into several successive steps, by first generating a suitable abstract model of the final output and then iteratively refining it. Dormans [4] describes a method using graph-rewriting systems to generate an abstract mission-model of the desired content prior to translation to a usable play space, and Karavolos et al. [9] provide an extension supporting mixed-initiative translation to two different game genres. Smith et al. [11] make use of a similar approach using ASP to generate high-level puzzle descriptions and then produce solvable playable instantiations of those puzzles.

3. Overview of Proposed System

The proposed system combines a constraint-based generation approach for valid content with a formal knowledge-representation input format, effecting content specification decoupling from generator implementation. End users extend a common base OWL ontol-

ogy (containing shared game concepts such as area, non-playable character and projectile) with entities and relationships specific to their domain of interest. Relevant portions of this are then translated to ASP and a valid production selected from the generated answer sets, with this process repeating either automatically or interactively (for mixed-initiative generation) until all necessary elements of the domain have been generated.

There is further potential for this approach to be used at runtime, using information from the current play session to provide responsive, dynamic content generation. Alternatively, the process could be run within an editor using existing content as input, for the purposes of constraint satisfaction verification as described by Boenn et al. [8].

3.1. Commercial Engine Integration

To support flexible content generation during design or at runtime, the system must be integrated with both the game engine and whatever editor tools are available. An early implementation integrates the monolithic ASP system *Clingo*[3] [14] with the commercial games development platform *Unreal Engine 4*[4] (UE4). Clingo is built as a shared library plug-in for UE4 and made accessible from the visual scripting language in order to load ASP files or code fragments, solve, and emit output in JSON to be parsed and processed by the individual game in a domain-appropriate manner.

3.2. Goals and Research Questions

The overall goal of the research is to demonstrate the feasibility and efficiency of using a general logic-based solver approach to generate a range of content definitions from semantic descriptions of the desired generative space. The primary motivation for this is the aim to decouple domain-specific information from the generation process, by developing suitable formats and patterns for the provision of high-level semantic domain data, and producing reusable generator components that can transform these specifications into concrete playable environments or other desired content. We intend to show the versatility of our system by generating sample artefacts of a range of kinds, and show effectiveness by evaluating them against the output of other respected generators. There is also potential for user-based expert evaluation of the system workflow by professional game designers familiar with UE4.

4. Combat Wave Generation

As an initial target, we aim to show that an appropriate logic-based solver approach can generate varied and challenging wave progressions for action/combat games, similar to the approach for puzzle games in Butler et al. [10]. Action/combat games often include competitive challenge modes consisting of a progression of combat waves as a test of skill and endurance, where a wave progression is an ordered succession of small groups (waves) of enemy characters which is traditionally hand-authored during development of the game. Authoring can be a complex process involving multiple rounds of testing, and may need to be repeated if the strengths of the enemies or player character change.

[3] http://potassco.sourceforge.net/ – Clingo and related tools, accessed 19 May 2016
[4] https://www.unrealengine.com/ – Unreal Engine 4, accessed 19 May 2016

Under the new system, progressions are generated using the Clingo integration described above, with a hand-authored ASP encoding containing designer-provided constraints relating to engine and player limitations. With reference to lines in the code fragments below, individual wave discriptions (2) consist of a composition of groups (1) of enemy types E and numbers C, and a total estimated difficulty D_total for the wave based on designer-specified approximate difficulties D for individual enemy groups. Similar rules produce waves of the form in (2), of 1-4 enemy types using designer-specified restrictions on wave composition, including a sameType(Ea,Eb) atom that prevents multiple inclusions of the same enemy in a wave and is also extended for multiple enemy types that fulfil the same role. The estimated difficulty D_total is used to compose waves into a progression of numbered spec(N,D,E) atoms (3), using constraints on maximum difficulty deltas Min and Max updated for each round. spec atoms are emitted for conversion to JSON output by the system.

```
1 group(g(E,C,D))
2 wave(w(D_total, e(g(Ea, Ca, Da), g(Eb, Cb, Db))))
3 1{ spec(N, D, E) : wave(w(D,E)), Min < D, D < Max }1 :- nxt(N, Min, Max).
```

Progression generation in this manner reduces the authorial burden on designers, and allows progressions to be easily re-generated in response to changes in combat or enemy balance. If generation is performed at runtime rather than during development, the generated progression could be different each time, reducing the effects of memorisation and encouraging player skill and responsiveness.

4.1. Extensions

At runtime, each successive wave spec could be selected as-needed based on information about the player's recent or overall performance such as total time or damage taken, to responsively tailor the experience to individual players' ability levels. The constraint-based system could also support more detailed player models such as the concept mastery approach described by Butler et al. [15], to guide the gradual introduction of new concepts such as additional weapons, abilities and enemies, and function as a tutorial experience for new players or a practice environment for mastery development.

5. Future Work

The wave progression generation approach demonstrates the use of ASP to generate content within a commercial game engine, however the ASP encoding used is still tightly coupled to the domain. Future work will include development of the OWL translation approach for ASP authoring (including investigation of suitable patterns for formal representation of content domains), and then production of a design space partitioning approach to guide decomposition of the problem into smaller iterative refinement steps.

Preliminary investigations were undertaken to investigate the application of OWL data provision and external computation for the generation of simple rogue-like dungeon levels from a basic ontology. More complex ontologies could be authored via domain-specific extension of a base ontology of common concepts, to be developed. This base ontology can then be paired with appropriate ASP modules designed to generate each included concept, and translation performed via composition of the appropriate modules with a re-usable skeleton problem encoding of common constraints and rules.

The base ontology can also guide separation of the generation problem into separate steps via structural hints and metadata. Each of the concepts within it will be part of a hierarchy representing increasing levels of detail, and can also contain information about generation prerequisites. Pre-processing as part of the ASP translation can use this to assemble separate problem encodings for each step, ensuring only relevant information is included in order to minimise time needed for solving.

Final evaluation will involve producing content from a variety of different domains to assess the generality of the system. It may also be appropriate to undertake user studies with designers to investigate the usability of the domain specification process and the utility of the output. Domains consisting largely of discrete, structured data are likely to be most well-suited to this approach, and a number of these domains have existing generators available which may be used for comparative evaluation. We hope to demonstrate a general ASP-based generation system via decoupled formal design space descriptions.

References

[1] Julian Togelius, Alex J. Champandard, Pier Luca Lanzi, Michael Mateas, Ana Paiva, Mike Preuss, and Kenneth O. Stanley. Procedural Content Generation: Goals, Challenges and Actionable Steps. In *Artificial and Computational Intelligence in Games*, volume 6 of *Dagstuhl Follow-Ups*, pages 61–75. 2013.

[2] Grigoris Antoniou and Frank Van Harmelen. Web ontology language: OWL. In *Handbook on Ontologies*, pages 67–92. Springer, 2004.

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

[4] Joris Dormans. Level design as model transformation: a strategy for automated content generation. In *Proc. 2nd International Workshop on Procedural Content Generation in Games*, page 2. ACM, 2011.

[5] Jassin Kessing, Tim Tutenel, and Rafael Bidarra. Designing semantic game worlds. In *Proc. 3rd Workshop on Procedural Content Generation in Games*, page 2. ACM, 2012.

[6] Xenija Neufeld, Sanaz Mostaghim, and Diego Perez-Liebana. Procedural level generation with Answer Set Programming for General Video Game playing. In *Computer Science and Electronic Engineering Conference (CEEC), 2015 7th*, pages 207–212. IEEE, 2015.

[7] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer Set Programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

[8] Georg Boenn, Martin Brain, Marina De Vos, and John Ffitch. Automatic music composition using answer set programming. *Theory and practice of logic programming*, 11(2-3):397–427, 2011.

[9] Daniël Karavolos, Anders Bouwer, and Rafael Bidarra. Mixed-initiative design of game levels: Integrating mission and space into level generation. In *Proceedings of the 10th International Conference on the Foundations of Digital Games*, 2015.

[10] Eric Butler, Adam M Smith, Yun-En Liu, and Zoran Popovic. A mixed-initiative tool for designing level progressions in games. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 377–386. ACM, 2013.

[11] Adam M Smith, Erik Andersen, Michael Mateas, and Zoran Popović. A case study of expressively constrainable level design automation tools for a puzzle game. In *Proceedings of the International Conference on the Foundations of Digital Games*, pages 156–163. ACM, 2012.

[12] Owen Cliffe. *Specifying and analysing institutions in multi-agent systems using answer set programming*. PhD thesis, University of Bath, 2007.

[13] Sarah Alice Gaggl, Lukas Schweizer, and Sebastian Rudolph. Bound your models! How to make OWL an ASP modeling language. In *Proceedings of IULP 2015*, 2015.

[14] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + control: Preliminary report. In *Technical Communications of the Thirtieth Internation Conference on Logic Programming*. ICLP, 2014.

[15] Eric Butler, Erik Andersen, Adam M Smith, Sumit Gulwani, Zoran Popović, and WA Redmond. Automatic game progression design through analysis of solution features. In *Proc. of the SIGCHI Conf. on Human Factors in Computing (CHI'2015)*, 2015.