**DOCTOR OF ENGINEERING (ENGD)**

# Procedural Constraint-based Generation for Game Development

Smith, Tristan

*Award date:*
2023

*Awarding institution:*
University of Bath

# Procedural Constraint-based Generation for Game Development

submitted by

Tristan A. E. Smith

for the degree of Doctor of Engineering

of the

University of Bath

Department of Computer Science

November 2020

# Summary

Game designers produce structured content for games that is informed by both hard gameplay requirements and design desiderata they must track and fulfil. Procedural content generation techniques can help, but are often inflexible, unwieldy or opaque; it can be difficult to make precise alterations or predict the effect of individual changes. We contribute a novel combination of approaches and develop the 'encode, evaluate, explore' paradigm as a method for defining and then refining generators' output. We build on design space sculpting literature to directly describe a desired generation domain, and integrate a quantitative visualisation approach to inform designers' alteration decisions. We use declarative constraint satisfaction, in the form of answer set solving, to directly describe a space of possible outputs in terms of the characteristics of desirable solutions.

We present applications of the initial 'encode' portion of the approach in two different domains, that represent interesting structured generation problems and are informed by industry needs. First an adaptable, parameterisable solution for automated generation (and regeneration) of varied and increasingly-demanding combat sequences for action/combat games, by composing fitting selections of enemies under provided constraints. Also, a tool for complex level layout specifications for action-adventure games, drawing on playability constraints that include structural and temporal 'lock-and-key' dependencies from dungeon generation literature. We implement the 'evaluate' phase via quantitative expressive range analysis, and show accessible visual characterisations of the output space of the latter generator system that facilitate visualisation of both iterations that 'explore' the formulation and also comparisons with the outputs of another generator.

By applying the methodological cycle of encode, evaluate, explore, we improve on existing approaches in two relevant domains. This dissertation shows that declarative constraint-based generation combined with clear visualisations and fast iteration provides game developers a novel, effective approach to produce structured content, and suggests multiple avenues of further research.

# Table of Contents

# List of Figures

# List of Tables

# List of Listings

# List of Abbreviations

**ASP — Answer Set Programming**; a constraint-based declarative programming approach for finding valid solutions to hard search problems encoded as logic programs (Chapter 3).

**DmC — *DmC: Devil May Cry***; the fifth entry in Capcom's Devil May Cry series, developed by Ninja Theory. An action-adventure game consisting of linear narrative levels containing combat and traversal challenges, building on Ninja Theory's signature mêlée combat style (Ninja Theory Ltd. (2013); *Definitive Edition*, (2015)).

**DSL — Domain-Specific Language**; a constrained custom language containing elements relevant to the domain of interest; frequently backed via the facility to directly transliterate to a more powerful/more general language, and developed in order to simplify application of the more powerful language to problems within the described domain.

**EngD — Engineering Doctorate**; a doctorate-level degree equivalent to a PhD that consists of a year of taught courses followed by at least three years of PhD-level research on placement with a company in industry.

**EPSRC — Engineering and Physical Sciences Research Council**; the UK's main agency for government funding for grants to undertake research and postgraduate degrees in engineering and the physical sciences.

**ERA — Expressive Range Analysis**; a quantitative technique for visualising the distribution of a generator's outputs over certain domain-specific metrics. Can be used to analyse the impacts of changes or compare two generators that work within the same domain. See Sec. 6.1.

**greybox**; a rough sketch of a playable space, typically using grey squares/cuboids to indicate the rough sizes and relative locations of areas within the space.

**JSON — JavaScript Object Notation**; a human-readable key-based open format for structured data, based on JavaScript's object syntax.

**lerp —*l*inearly int*erp*olate**; calculate an intermediate value between two given values according to relative 'distance' from each in some space. If the space is physical, then this is a point along the straight line between the two points; if the values are separated by time

then it is the intermediate value that will reach the second value under constant velocity of change, given the time already elapsed.

**OWL** — the **Web Ontology Language**; a knowledge representation language for describing both the schema and instances of entities and relationships within a domain.

**PCG** — **Procedural Content Generation**; an automated approach for constructing digital artefacts. In a widely cited survey paper, Togelius et al. (2011a) define PCG as *the algorithmical creation of game content with limited or indirect user input*.

**RPG** — **Role-Playing Game**; a game genre in which players embody a specific viewpoint character within an immersive digital setting, and make personal choices about progression, exploration and frequently combat.

**RTS** — **Real-Time Strategy**; a game genre in which players assume control of a whole faction within a combat setting, and make high-level strategic decisions about construction, recruitment and army/unit manoeuvres.

**UE3** — **Unreal Engine 3**; a 3D game engine developed and distributed by Epic Games, 2006.

**UE4** — **Unreal Engine 4**; a widely-used modern 3D game engine, editor and associated ecosystem of launcher, marketplace, tools and workflows developed and distributed by Epic Games; released 2014.

**UGC** — **User-Generated Content**; content that users have generated within game systems that explicitly allow them to do so. These range from fully manual editors (e.g. Super Mario Maker) to highly assistive mixed-initiative editors (e.g. the Spore Creature Creator) in the 'casual creator' tradition (Compton and Mateas, 2015).

**WBC** — **Wave-Based Combat**; in-game combat experience consisting of successive 'waves' where each is formed of one or more groups of one or more of a kind of enemy. This is typically distinct from the successive groups of enemies found throughout levels in games containing combat as the waves simply appear, either on a fixed timer interval or automatically shortly after the preceding wave is defeated.

**WFC** — **Wave Function Collapse**; method for deriving distribution and connectivity information from minimal exemplars and then tiling valid arrangements over the specified region.

**ZDG** — the **Zelda Dungeon Generator**; a graph-grammar-based approach for generating dungeons for a Zelda-like game; (Lavender and Thompson, 2015).

# Acknowledgements

This thesis would not exist without the patience and dedicated support of my parents Kate and Arthur, and of my academic supervisor, Dr Julian Padget. I also owe a debt of gratitude to so many people who have supported me throughout in so many different ways, in particular Jack, Callum, Jon and Court, who have seen the entire journey.

I have been fortunate to meet and chat with many of the researchers upon whose shoulders this work stands – whose questions, insights and public passion for our shared academic interest helped to nurture and sustain my own. Assortedly: Dr Mike Cook, Dr Adam Smith, Dr Kate Compton, Dr Gillian Smith, Dr Anne Sullivan, Dr Julian Togelius, Dr Max Kreminski, Dr Isaac Karth, Dr Joris Dormans, Dr Daniel Karavolos, Dr Stella Mazeika, Dr Antonios Liapis, Dr Tommy Thompson, and Alex Champandard & Petra Champandard-Pail, whose nucl.ai conference led directly to my ongoing career.

I am grateful for pastoral and academic support from the staff at the CDE, especially Sarah Hayward, Sarah Parry and Dr Leon Watts, and I want to also acknowledge invaluable additional professional assistance from the therapists and counsellors that have helped me along the way: Polly Wellby, Nicola Peacock, Paul Johnson, Jo Walmsley-Moore and Donna Jelley.

In Cambridge Laurie, Fabian, Oly & Em, Fran, Alex, James, Joe and Sky provided social support and occasional technical tips during my research. Later Troy and Chris gave me a place to stay (and more importantly, with Henry, Ro and Rey, a place to stay sane) through lockdown and writeup, and then Sarah kept me fed. Peter, Neal, Tony, Jim, Max, Mike, Stephen & Adam gave advice won from experience, and have helped more than they know. With respect to the ninety-ninety rule I'm also very grateful to my understanding manager Dr Tim Gosling, and for support on the final stretch from Jay, Kate and Jevon. And again throughout: my parents.

We thankfully acknowledge the support of the Engineering and Physical Sciences Research Council (EPSRC) through the Centre for Digital Entertainment (CDE, University of Bath), under grant reference EP/G037736/1. We would also like to thank Beck Lavender for permission to reproduce figures from her work for comparison (Lavender (2016), Chapter 6), and Mark Brown for developing the Boss Key dungeon graphing approach and making the visual resources freely available (Brown (2017), Chapter 5). We also thank our anonymous paper reviewers, in particular for the recommendation of corner plots (Foreman-Mackey (2016), Chapter 6) as an appropriate visualisation for expressive range metrics across more than two dimensions.

# Chapter 1

# Introduction

Procedural Content Generation (PCG) is the production of game content using automated procedures and limited designer input; however common approaches are either highly bespoke and difficult to re-use, or have difficulty generating content that respects hard gameplay constraints (Togelius et al., 2013a). Our research builds on the design space descriptions approach developed by Smith and Mateas (2011) in order to provide controllable constraint-driven generation applicable to extant industry needs in two distinct domains. We also demonstrate comparability of our approach using a popular evaluation technique, Expressive Range Analysis (ERA) (Smith and Whitehead, 2010). In this chapter we introduce the field and motivation of the research, with initial summaries of the key concepts, and an explanation of the context in which the work was undertaken. We present an overview of the research contributions of the work, and conclude with an outline of the remainder of the document.

This thesis describes our research into providing constrained procedural content generation via answer set programming. Many opportunities for automated design assistance in the field of commercial video game development come with associated playability and design constraints, and we show that constraint-based generation is applicable to two of these contexts.

In this section, we lay out the initial motivation for the research. Games and interactive entertainment are a large and growing market segment, worth £7.16bn in 2021 in the UK alone[1]. Frequently, game developers both large and small make use of PCG techniques in order augment the process of authoring content for these games. A wide range of approaches have been developed across both industry and academic literature, however it is commonly the case that many generators are developed for a single game context or narrow domain and the bespoke nature of these approaches can hinder future re-use. Current PCG systems are often highly bespoke and encode implicit assumptions about the desired content in the design or implementation of the generator, which can also make it difficult to predict precisely their expressive capabilities — an exploration of the existing literature is presented in Chapter 2. Though PCG approaches can and have been used for predominantly cosmetic content (see e.g. Speedtree (Interactive Data Visualization, Inc., 2002)), there are also opportunities to gen-

[1] `https://ukie.org.uk/consumer-games-market-valuation-2021` — accessed 14/11/2022.

erate 'functional' content such as playable spaces or useable items, intended to form part of the core player interactions. These forms of generated content must satisfy hard playability constraints in order to avoid breaking the play experience. However, this requires that the designer-intended playability constraints must be explicitly specified, introducing an authoring burden. Our research makes use of Answer Set Programming (ASP, Chapter 3) and the 'design space' sculpting approach suggested by Smith and Mateas (2011) to provide a mechanism for industry designers to specify and adjust elements of these constraints, and generate content accordingly. We discuss our application of this approach to an initial industry-relevant problem (combat wave progression generation; Chapter 4) and then further application to a more complex problem (action-adventure level greybox generation modelled on common dungeon design patterns; Chapter 5). A key acceptance criteria for industry usage of PCG is observability: the idea that not only should the generated content be validly playable, but that designers also need the ability to introspect on what kinds of content the generator commonly produces, and how changes to the inputs affect the character of the outputs. The behaviour and output of a generator can be challenging to alter as desired without an understanding of the intention or implementation of particular features. An approach for generator evaluation and investigation that has shown significant promise is Smith and Whitehead's (2010) ERA, which we apply to our ASP-driven greybox generator in Chapter 6 to help assess the generative space of our implemented approach, and provide comparison with an alternative approach for generating content within the same domain (Lavender and Thompson, 2016). Finally, we suggest that while the research presented in this thesis has advanced the state of the art in these areas, it has also uncovered several promising further questions and opportunities for continuation of the research (Chapter 7).

In the remainder of this chapter we provide further detail on the key concepts and background context for the research, followed by a summary of the research contributions and an overview of the rest of the thesis.

## 1.1 Procedural Content Generation

In this section, we place our work in context within the field of PCG research. We discuss a commonly-accepted definition and the attention the field has received both in academia and as a successfully-applied technique in commercial games. In a summary of the research area, Togelius et al. present a range of suggestions for the field's Grand Goals for the future including the challenge of providing *general content generation*, which we explain and illustrate with a motivating example from science fiction. We explain the background of the quote and use it to provide an intuition of the gap between the state of the art and the 'multi-level multi-content PCG' Goal, and explicitly position our work as a step towards closing that gap.

> "Procedural content generation (PCG) refers to the algorithmic generation of game content with limited or no human contribution."
>
> Togelius et al., *Procedural Content Generation: Goals, Challenges and Actionable Steps.* (2013a)

Togelius et al. (2013a) present a widely-cited definition for PCG (above), and approaches of this sort are increasingly used by games companies to provide diverse and varied content. Research into PCG for games is a broad topic, however many implemented systems are highly bespoke and tightly coupled to either the game or the content they were initially designed for. This makes it difficult for developers unfamiliar with the workings of a generator to alter its output or repurpose it for other games or content types, limiting opportunities for the reuse of code and systems and the associated benefits.

One of the explicit goals proposed for the field is the production of 'multi-level, multi-content PCG'; i.e. approaches that are capable of producing more that one kind of content for a given game, where elements of the generation process are able to be informed by prior decisions relating to other content generated by the same process. Examples given include map and quests generated together, though the overall goal proposes production of the entire structure of supporting functional content, including enemy kinds, lore, dialogue and cosmetic decoration. They suggest that progress towards the field's goals may be achieved by work aimed at tackling some of the field's Challenges, one of which being the production of *general content generators*; generic '[plug-and-play] PCG systems that could be used without further development to generate for example levels or characters for a new game' (Togelius et al., 2013a). Their intent is that such a general system should be usable to generate multiple kinds of content only by altering the inputs and without requiring any additional development effort. The aim is that effort invested in producing a procedural generator that is not tightly coupled to a specific domain becomes reusable in a way akin to other forms of middleware and so can be amortised across multiple purposes; in addition any further improvements can be more easily cross-implemented to anywhere the same approach is used.

Togelius et al. describe 'create[ing] a game no-one has played before [...] at the press of a button' as part of one of the related field goals: 'generating complete games'. This is a popular concept in science fiction, from Star Trek's Holodeck (Roddenberry, 1987) to the custom virtual adventures in Clarke's *The City and the Stars* (Clarke, 1956), to the Free Play in Card's *Ender's Game*, as described below.

> "It was private study time, and Ender was doing Free Play. It was a shifting, crazy kind of game in which the school computer kept bringing up new things, building a maze that you could explore. You could go back to events that you liked, for a while; if you left them alone too long, they disappeared and something else took its place. Sometimes funny things. Sometimes exciting, and he had to be quick to stay alive."
>
> Orson Scott Card, *Ender's Game* (1985)

The fictional system described in this novel is shown to draw from cultural allusions (The Gi-

ant & the Beanstalk, literal cat-and-mouse games, the Red Queen's Games from Alice Through the Looking-Glass) and to be highly responsive to the player's implicitly-expressed preferences, developing on themes and areas that receive attention and deliberately challenging the player. This implies a generative system that is not only equipped with constraints relating to both satisfying game design and also cultural semantics and stories, but is additionally capable of performing (and altering its outputs in response to) some level of player psychological evaluation. Though contemporary systems have not reached this level of sophistication, each of these fields is the subject of active research. Current PCG research already overlaps strongly with the fields of player modelling (e.g. Bicho and Martinho, 2018) and the investigation of semantics within games and generation (e.g. Tutenel et al., 2009b, and Sec. 7.2.3) which are steps towards this possible future, and our own work is informed by these areas and intends to present a similar step towards the direction of Togelius et al.'s Goals.

In this thesis, we discuss an approach to PCG that decouples the description of the space of desired content from the techniques used to generate it; related to the *design space descriptions* work by Smith et. al. Smith and Mateas (2011). We hope to provide a step towards one of the 'grand goals' of PCG research suggested in an overview of the field Togelius et al. (2013a) — specifically, an approach for *Multi-level, Multi-content PCG* — in the context of providing assistive approaches for the development of commercial games. We present two investigations that use ASP to generate constrained content for commercially-relevant scenarios, and describe our published work on evaluating one of these approaches in comparison to an alternative technique. We aim to further the development of plug-and-play content generation (Togelius et al.'s concrete research challenge: *General Content Generators*), by allowing designers to intentionally specify the requirements for content rather than the underlying approach that should be taken to produce it.

There are 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 build on prior work in the field to contribute to an approach for 'content-agnostic' generation — 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*. The system produces valid descriptions of instances of the designer-specified content, which could in principle vary from level layouts to branching narrative skeletons to complete descriptions of the gameplay environment. In this thesis, we present two applications of the ASP-based PCG approach to existing problems in the game design process. Through our connection to industry we have also been able to develop systems that convert these specifications into playable artefacts, for the purposes of comparison and testing. The development of a general content generator is described as an open research challenge in PCG in a recent overview of the field Togelius et al. (2013a), and potentially a step towards the 'grand goals' of

developing *Multi-level, Multi-content PCG* and *Generating Complete Games*.

## 1.2 Commercial game development

Commercial games make use of PCG approaches to augment manual development in a range of ways. Early games used deterministic expansion of an initial seed according to a well-defined algorithm as a form of content compression; and this approach lives on in the cosmetic applications by environment artists for realistically 'scattering' assets according to predefined rules, and improving immersion in fictional spaces by providing procedural variation that doesn't affect gameplay directly (e.g. using Speedtree (2002), Houdini (1996), generated textures and other cosmetic aspects). On the other end of the spectrum some games build core features or the entire game direction around the use and exploration of 'functional' generated content (Borderlands, No Man's Sky, for more see Sec. 2.4). Where procedural aspects can affect gameplay it's frequently important that certain playability constraints such as traversability remain true or valid, though some game designs avoid this issue this through empowering the player to avoid or ignore problematic generation outcomes (*Spelunky* (Yu, 2016) is frequently cited as an example of this approach).

The process of commercial game development includes many tasks that could be simplified or partially automated using procedural techniques, however the rapid pace of typical development frequently does not allow time for detailed investigation of how these techniques could be productively applied. The EngD programme (Sec. 1.3, below) is designed to allow scope for productive research into industrially-relevant problems. We have applied a technique under active academic investigation to two largely novel domains: wave-based combat progressions, and action-adventure dungeon generation. These domains are contained subfields of larger design problems in games, and so our work is both applicable to the specific commercial contexts we tackle and a step towards providing generative approaches for the more general design area. As detailed in Sec. 4.1.1, the problem of wave-based combat progression generation is a subset of the challenges inherent in generating a satisfying and instructive combat progression for an entire game, and in Sec. 4.4 we suggest additional research that can help to close that gap. Similarly, our approach to generating self-contained dungeon levels for action adventure games as detailed in Chapter 5 is a step towards the constraint-based generation of entire action-adventure games overall, as many of the fundamental structure challenges are similar at both scales. In both cases we have shown promising results that are comparable to either designer-provided hand-crafted specimens or the output of an alternative procedural approach; integrated our systems with commercial software to demonstrate applicability of the concept; and detailed the ancillary benefits of our technique and how it may be developed further.

## 1.3 The EngD context

In this section we provide an introduction to the motivations, format and structure for the research that led to this thesis. The work was undertaken as part of an Engineering Doctorate

(EngD) programme, an Engineering and Physical Sciences Research Council (EPSRC)-funded doctoral training course leading to a PhD-level qualification. The course combined a year of academic taught components (at the Centre for Digital Entertainment in the University of Bath, 2013-14) with three and a half years of research projects whilst on industrial placement (2014-18). Final assessment is via defended thesis, however in addition to showing an academic contribution to the field the outputs of EngD programmes are intended to have relevance to the industry partner, whose needs may change over time according to project lifecycles and external factors.

Students are therefore embedded on placement with a relevant industry partner, working on research that is topical for their partner company. The sponsor for this project was Ninja Theory ltd.[2], a games studio based in Cambridge U.K. that develops a mixture of first- and third-party games and is now wholly owned by Microsoft.

### 1.3.1 Ninja Theory

Ninja Theory, the partner company with which this work was produced, is a UK Cambridge-based games studio that has historically focused on producing third-person mêlée action/combat games, including *Heavenly Sword* (2007), *Enslaved: Odyssey to the West* (2010), *DmC: Devil May Cry* (2013; *Definitive Edition* 2015) and *Hellblade: Senua's Sacrifice* (2017). They have also contributed to *Disney Infinity 2.0 & 3.0* (2014, 2015), and developed a 'hyper-reality experience' *Nicodemus: Demon of Envanishment* (2018), which all build on another core studio strength: semi-linear levels containing puzzles.


**Figure 1**: A combat encounter in DmC with two different enemy types visible: a shielded and slow-moving Greater Stygian, and two chainsaw-wielding Ravagers, capable of quick charges.

[2] `https://ninjatheory.com/about/` — accessed 15/11/2022

Ninja Theory games typically provide the player with a lengthy narrative-driven 'story' mode, consisting of a set of designer-authored environments and challenges which the player must overcome in order to progress. Though some puzzle and exploration challenges are occasionally present, the focus of most of the games is on skill-based mêléecombat. This gameplay focus is supported by a range of available skills and weapons, and a variety of opponent characters to defeat — each with their own special skills and weaknesses. The initial narrative mode of these games presents the player with few options and assumes minimal pre-existing knowledge. During the course of completion of the games' main narrative, additional skills and enemies are introduced in order to provide increasing challenge and complexity, and typically this happens gradually to allow the player opportunities to learn how to use or counter these concepts. This progression of increasing ability and challenge is intended to facilitate the player's mastery of the game systems, and is generally accompanied by an explicit scoring/ranking system to encourage repeat play and self-improvement.

A common additional feature of many action-adventure games (including some produced by Ninja Theory) is an alternate arena-style combat challenge mode, where the player is presented with successive 'waves' of enemies — small groups with varied compositions, where every enemy in the wave must be defeated before the next wave will arrive. In order to complete these Wave-Based Combat (WBC) challenges, the player must typically complete all of the waves of enemies without running out of health, with bonus health awarded for defeating certain enemies. Optionally, a time limit may also be present, with time extensions awarded for defeating enemies, and bonus time for defeating enemies with 'style', as determined by the game's 'style system', which attempts to measure the skill and variety of the player's combat.

DmC (Capcom, 2013) is a third-person action/combat game developed by Ninja Theory that contains two variations on this challenge mode feature, one for each of the two playable characters once the respective storyline has been completed. The feature is known as the 'Bloody Palace', and presents a variety of combat environments and 60 or 101 'rounds', each consisting of either a repeat appearance of one of the 'boss' enemy battles from the main storyline (one every 20 rounds), or up to two waves of enemies. An open problem is the difficulty of authoring content in the form of wave progressions for these features — though the overall difficulty of successive rounds generally increases, it does not do so predictably or linearly, in order to provide a varied and challenging progression of tension throughout the combat. In addition, the interaction between fighting styles and abilities of various combinations of enemy characters can make it difficult to precisely predict the expected difficulty of a given wave composition: a typical development approach is for an experienced designer to produce a complete wave progression specification, and then hand this off to an experienced tester to evaluate and provide feedback. Though this process empirically works, and was used to produce both iterations of Bloody Palace in *DmC*, it has a number of drawbacks in that it takes a non-trivial degree of labour to both produce and test possible progressions, and must be at least partially repeated to perform re-balancing if any combat attributes of the player or enemy characters are altered. Typically this labour is (and can only be) performed by senior designers and testers working according to personal heuristics developed through long experience.

*Heavenly Sword* (2007) was Ninja Theory's first game as a studio, a combat-based 'hack and slash' action adventure game focused on martial arts-styled mêlée combat with three distinct weapon forms, and a secondary character with an alternate ranged combat approach. This was followed by *Enslaved: Odyssey to the West* (2010), a sci-fi retelling of the classical Chinese novel Journey to the West — again focusing on third-person combat in a narrative structure, but with the addition of significant traversal challenges and some puzzles.

*DmC* (2013), was a reboot of Capcom's Devil May Cry franchise; Ninja Theory's first foray into melding a pre-existing intellectual property with their approach to varied combat and platforming challenges. A free downloadable expansion to the game contained a wave-based 'arena' combat challenge, as detailed above, and another contained a second, shorter narrative experience with new weapons, abilities and enemies. *DmC: Definitive Edition* (2015) is a re-mastered version of the game for eighth-generation consoles and included a second, shorter variation on the wave-based combat challenge concept using the additional content from the earlier downloadable expansion. *Hellblade: Senua's Sacrifice* (2018) is Ninja Theory's most recent mêlée action-adventure hack-and-slash title; which contains psychological themes and puzzles and was developed following an 'independent AAA' game development approach, aiming for the high level of polish typical of AAA games but with a shorter playtime, reduced team and consequent smaller budget.

Ninja Theory have also developed several other titles that explore and expand on the studio's core competencies, including: *Fightback* (2013), a free-to-play mobile multiplayer game; *Disney Infinity 2.0: Marvel Super Heroes* (2014), a toys-to-life action-adventure game, part of the larger Disney Infinity franchise; *Disney Infinity 3.0: Twilight of the Republic* (2015), a similar themed game with focus on the lightsaber mêlée combat of the Sith and Jedi characters; *DEXED* (2016), a self published VR game and *Nicodemus: Demon of Envanishment* (2018), a installation-based VR horror experience. In late 2017 the studio made public the details of its cancelled game *Razer*[3], a massively-multiplayer persistent game that would have had procedurally generated missions and scenarios but did not make it past preproduction after initial strong interest from publishers was withdrawn when a competitor was announced[4].

Ninja Theory have a clear interest in the efficient production of action-adventure games focusing on mêlée combat. The work in this doctorate is applicable to real commercial design issues encountered by Ninja Theory and other companies engaged in the practice of games development.

## 1.4 Research contributions

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 designer-guided high-level descriptions of the desired generative space. The primary motivation is the

[3]https://www.hellblade.com/unseen-ninja-theory-razer/ — accessed 27 November 2020

[4]`https://www.eurogamer.net/ninja-theory-details-its-cancelled-action-game-razer` — accessed 15 November 2022

aim to decouple domain-specific information from the generation process. We approach this by developing suitable formats and patterns for the provision of high-level semantic domain data, and producing versatile plug-and-play generator components that can transform these specifications into concrete playable environments or other desired functional content. We show the versatility of our system by generating sample artefacts of a range of kinds, and show effectiveness by evaluation against the output of other respected generators, using ERA as described by Horn et al. (2014).

As an initial target, we show that general logic-based solver modules can be used with a suitable knowledge base of ASP-encoded constraints and relationships to generate satisfying wave-based combat progressions and action-adventure dungeon level greyboxes. We present comparisons with existing hand-authored outputs in the WBC domain, and with existing generator outputs in the dungeon domain. There is also potential for user-based expert evaluation of the system workflow by professional game designers familiar with UE4, who would be able to give feedback on the subjective usability of the system, as presented in Sec. 7.2.1 and the Appendices.

The presented research consists of four main tasks repeated across two industrially-relevant domains: 1) the investigation of suitable formats and patterns for the provision of high-level semantic data about the desired properties of playable design 'spaces', 2) the development and refinement of generator components that can transform these semantic specifications into instances that describe concrete playable content, 3) the integration of these specifications and components with widely used industry tools to support developer workflows, and 4) evaluation via comparison with existing approaches in the relevant domains.

Ultimately, it should be possible for designers to supplement a provided ontology of base concepts with refinements specifying classes and relationships within the desired content, and then use the solver system and this augmented ontology to automatically generate or verify and evaluate content that fits their specifications — we present work that contributes to development towards this goal, and discuss promising next steps in Chapter 7.

In general, generator design choices may implicitly constrain the space of possible outputs, and these internal biases and limitations may easily go unobserved without formal evaluation designed to assess the generator's expressive range (Smith and Whitehead, 2010).

The scope and versatility of our system implementations is shown by generating sample artefacts and evaluating them against the output of other pre-existing generators, as in Chapter 6. This is an active area of research with ongoing establishment and refinement of both domain-specific and 'portable' evaluation metrics, and we have contributed to this work by publishing the outcomes of our dungeon generation work and evaluation (Smith, Padget and Vidler, 2018).

PCG systems for new contexts are frequently bespoke and tightly coupled to each new domain, and therefore learnings and code are difficult to re-use. The behaviour of a generator can generally only be reliably modified by someone who understands how the code works and why. The expressive range of a particular generator can often be difficult to precisely quantify, as many of the constraints that define the shape of the output space are implicitly encoded

within the design or implementation of the generator itself. Development of a more content- and algorithm-agnostic generator as described is a step towards the ambition of being able to produce 'plug-and-play' PCG middleware for a range of games and content types, which is described as one of the goals of the academic field (Togelius et al., 2013a). In addition, effective methods for improving the production of game content are of interest to our industry partner and the wider industry at large, with ease of evaluation and direct visibility of the effects of changes being an important aspect of the acceptance criteria.

The availability of a truly general, modular content generator would reduce the time spent developing bespoke solutions for particular applications or domains. The same proven code base could be reused across multiple projects, and familiarity with the system would transferable into each new context. In addition, any improvements to the capabilities or the efficiency of the system could provide benefit across all projects that used it. It is equally important to end-users that the outputs of such systems be observably correct in specific ways, and that the effects of changes be legible. Both developers and users of PCG systems would benefit from the existence of generators that use a common, repeatable, inspectable approach across content domains. This research intends to provide a step towards developing reusable, meaningful content generation.

## 1.5 Thesis overview and declarations

The material presented here for examination for the award of a higher degree by research has not been incorporated into a submission for another degree. I am the author of this thesis, and the work described therein was carried out by myself personally except where explicitly otherwise credited (certain illustrative figures and a table in Chapters 2 & 7; permitted reproductions of the analyses of Lavender for comparison purposes in Chapter 6).

This thesis contains portions from two original papers published in two peer-reviewed conferences. The contents of the thesis and the two papers used therein were the principal responsibility of Tristan Smith (formerly known as Thomas Smith at the time of previous publications), working within the University of Bath and Ninja Theory Ltd. under the supervision of Dr Julian Padget (University of Bath) and Andrew Vidler (Ninja Theory).

Parts of the content of Chapter 4 were published as part of a submission to the Starting AI Researcher Symposium (STAIRS) at the European Conference on AI 2016 (Smith, Padget and Vidler, 2016).

Portions of Chapters 5 and 6 were published in the 9th Workshop on Procedural Content Generation at Foundations of Digital Games 2018 (Smith, Padget and Vidler, 2018).

In Chapter 2 we present an overview of relevant literature, covering the definition of PCG, summaries of a range of common or well-established approaches including ASP, an overview of interactive techniques and the intersection with the field of User-Generated Content (UGC), and a introduction to some of the field's existing evaluation methods including ERA.

Chapter 3 provides an initial grounding in the fundamental facts relating to ASP, including an overview of syntax and practical usage. A more thorough review of specifically ASP-related

PCG literature is supplied, with discussion of the 'design space approach' developed by Smith and Mateas (2011).

Our application of ASP-based generative methods to a concrete game development problem is detailed in Chapter 4: Wave-based Combat. We provide a thorough explanation of the problem and its definitions, introduce and justify our approach via illustration of the ease of reformulation, and suggest how the work could support additional forms of challenge experiences.

Our further application of ASP to problems encountered by industry is considered in Chapter 5: Constrained Dungeon Design, where we discuss another potentially complex constrained domain. We repurpose an existing visual description language as an abstraction suitable for structured generation, and detail the evolution of our approach. We briefly describe work undertaken to integrate this system with a commercial editor, and review the opportunities for further development that the work has exposed.

In Chapter 6 we present and apply an evaluation approach for procedural generators (ERA) to the system described in the previous chapter. We make use of this evaluative approach to draw comparisons with another generator developed for the same domain, and also to illustrate the effects of certain changes to our formulation, demonstrating the ease with which the output space of our generator may be deliberately shaped.

Finally, Chapter 7 presents a summary of our contributions, discusses the conclusions drawn in earlier chapters and shows how these lead naturally to a range of promising further possibilities and additional potential applications of these techniques.

# Chapter 2

# Procedural Content Generation for Games

In this section, we explain general terms relating to the field of Procedural Content Generation (PCG) for games research, give an overview of the areas covered in this thesis, and review the related and prior work in the area. We draw upon advances detailed in related work in a number of areas: the development and evaluation of PCG systems, and the use of Answer Set Programming (ASP) for PCG. The research spans a range of existing fields of study, and attempts to draw upon the best practices and appropriate techniques developed in each. In this chapter, we present an introduction to contemporary study and methods in procedural content generation, with particular attention to suitable methods for evaluation of research in these areas. Then we cover a constraint-based approach for declarative problem-solving, and discuss its successful application as a tool for content generation in various forms. Finally we give an overview of a specific technique applied by some existing implemented generators as a means of reducing task complexity and increasing designer control over output.

## 2.1 Procedural Content Generation motivation

In this section we examine the motivations behind the research and development of PCG approaches. We look at the many varieties of procedural generation systems that exist, and the areas in which PCG blurs the lines with neighbouring disciplines. We also cover a number of PCG systems implemented in commercially released games.

As introduced in Sec. 1.1, "PCG is the algorithmic creation of game content with limited or indirect user input" (Togelius et al., 2011a). This definition deliberately omits specification of whether this should happen at run-time or during development, whether the process should be random, or deterministic, or responsive to external data such as a player model, and any details about purpose or the nature of the algorithm used. The broad terms of the definition reflect the breadth of approaches taken to generating content (Sec. 2.2), and the range of motivations for

which PCG is used in games. (In order to account for certain inadequacies of the terminology, a broader field definition 'Generative Methods' has also been proposed by Compton, Osborn and Mateas (2013).)

Included amongst the various motivations for using and improving PCG techniques are the desires for increased variety, solutions to problems of scale, use of PCG as an aesthetic, or as a component for user-generated content systems (Smith, 2014). Originally, PCG approaches were tools used by developers in order to expedite the process of populating game worlds in the presence of limited hardware. Increasingly however, PCG systems are sufficiently performant to be used during runtime, and therefore able to use information gathered about the player's actions as input, leading to a new class of PCG-based game designs.

> "[PCG] isn't a *substitute* for designing content. It's a *way* of designing content"
>
> Emily Short, (2016)

As a non-exhaustive list of commonly-expressed motivations: (i) reduction of content production costs, (ii) increased replayability of content, (iii) increased content variance (iv) augmentation of designer creativity (v) exploratory practice (vi) exploration of 'meaning' of content, or creation; discussed further in (Smith, 2014).

Given the variety of uses for PCG, there are an equal or greater variety of implemented approaches. In the rest of this chapter, we examine the literature relating to PCG used either within games, or as part of the process of game development. We also interrogate an emerging area of the research concerning methods for evaluating individual systems' outputs, and facilitating comparisons between different generators that address the same domain.

## 2.2 Approaches to PCG

A comparatively recent survey of the field undertaken by Hendrikx et al. (2013) attempts to categorise both the classes of content that are currently generated (both in literature and commercially, see Table 2.1), and the range of systems used to generate these content types, including 'constraint satisfaction and planning' (the focus of this work) and 'genetic algorithms' (surveyed in more detail by Togelius et al. (2011b)). They present a taxonomy of generation approaches, and map between methods and content layers, with suggestions for under-explored areas and recommendations for future research. The survey is intentionally light on technical details, and instead aims to provide observations about the state of research in the field and implementations in commercial games.

As detailed in Hendrikx et al. (2013), procedural generators exist for many types and 'levels' of content, however few generators attempt to generate more than one kind. Some of these existing generators may serve as suitable points of comparison for domain-specific evaluation of the system, as above, and by comparing the system to multiple generators across multiple content types, a degree of generality may be demonstrated. Absolute domain-generality would be hard to quantify, and therefore hard to evaluate, but generating content across each or most

of the lower four levels of the content taxonomy in Table. 2.1 would indicate an improvement on the current state-of-the-art.

| Content class | Content type examples |
|---|---|
| Derived Content | News and Broadcasts, Leaderboards |
| Game Design | System Design, World Design |
| Game Scenarios | Puzzles, Storyboards, Story, Levels |
| Game Systems | Ecosystems, Road Networks, Urban Environments, Entity Behaviour |
| Game Space | Indoor/Outdoor Maps, Bodies of Water |
| Game Bits | Textures, Sound, Vegetation, Buildings, Behaviour, Fire, Water, Stone & Clouds |

Table 2.1: Taxonomy of game content types provided by Hendrikx et al. (2013).

Another survey, performed by Smelik et al. (2014) looks exclusively at the areas of the field concerned with the procedural modelling of components of 'virtual worlds' (not specifically games), and is written for an audience in the field of graphics research in order to communicate recent advances in the field of PCG research between domains. In comparison to the method-based or content-based grouping approach taken by Hendrikx et al. (2013), the authors consider generative approaches grouped by the semantic kind of the content generated, such as vegetation, building, terrain or city, and provide individual overviews of domain-specific papers covering approaches to generating each feature.

The popularity of PCG as a research domain has also led to the collaborative production of an instructional textbook by Shaker, Togelius and Nelson (2016). Originally developed for a university course on PCG, it aims to provide an introduction to the field and an overview of the major concepts and areas of research. An open access version is available online[1]. Each chapter focuses on a particular combination of approach and content type, and gives a high-level overview of how that approach works and may be implemented for that content type, combined with a substantial bibliography for research in that area.

Some researchers draw explicit links between the methods used in traditional procedural content generation research and those used in other similar fields such as algorithmic music, parametric architecture or generative grammars for computation linguistics (Compton, Osborn and Mateas, 2013). They also note the potentially problematic term 'content' in the usual appellation 'procedural content generation' — as the output of several contemporary generative systems (story structure, enemy behaviours, rules systems) is only notionally 'content'. On these points they argue that the field as a whole might more accurately be considered as the study of 'generative methods', with the traditional areas of PCG research most closely mapping to the study of generative methods *for games*. We note the increased accuracy of this terminology, but maintain the traditional usage of 'PCG' throughout the rest of the paper merely for convenience when referencing existing research.

There are a range of common approaches to PCG, as covered in Shaker, Togelius and

[1] `http://pcgbook.com` — accessed 27 November 2020

Nelson (2016) and the taxonomy presented in Hendrikx et al. (2013). Of particular note are generative grammars, search-based PCG, and constraint-based systems, for their widespread representation in contemporary literature. A brief overview of each of these techniques is presented next, and constraint-based systems (specifically, ASP and selected extensions) are described in further detail in following sections (Sec. 3.3).

### 2.2.1 Generative grammars

In this section we introduce the concept of generative grammars: sets of production rules that implicitly define spaces of content. After defining the terms and giving basic examples of the concept, we introduce some of the advanced kinds of grammar, and signpost the following paragraphs where we discuss academic uses of them. Initially we address basic uses of grammars and use of simple shape grammars for decorative purposes - mainly vegetation and building facades. Then we cover more advanced uses of spatial grammars for playable spaces, including dungeons; then multistage approaches: generation of mission graphs guiding playable spaces. Finally we mention some more tangential applications of grammars to the content generation problem, such as Tracery or SpeedRock (Compton, Filstrup and Mateas, 2014; Dart, De Rossi and Togelius, 2011).

'Generative grammars', also known as *Lindenmeyer systems* or simply *L-systems*, are formalised collections of (potentially recursive) rules that lead to the production of comparatively simple structured output. Grammars start with a 'root' *non-terminal symbol*, and rules define the replacement of non-terminal symbols with one of a set of productions, which are composed of further non-terminal symbols (which will be later expanded in turn) and/or *terminal symbols*, which are elements of the final output. The grammar `b: sb | s`, `s: 0 | 1`, with *non-terminals* `b` and `s`, *terminals* 0 and 1, when given root `b`, may be expanded to produce binary strings of arbitrary length.

The produced output of structured terminal symbols may be treated as an encoding of a particular configuration for the desired content, and transformed in a domain-specific manner according to a one-to-one mapping between symbol and content component (Trescak, Esteva and Rodriguez, 2010). This basic approach however is only suitable for linear productions, and so often graph- or spatial-grammars are defined, where rules encode additional information about the relations between output terminal symbols. This method is suited for generating a wide array of branching content types, from literal in-game trees (e.g. the widely-used commercial system SpeedTree (Interactive Data Visualization, Inc., 2002)) to tree-like acyclic graph structures representing dungeons (e.g. Lavender and Thompson, 2015, see Sec. 6.2.2) or choice-based narrative outlines (e.g. Dormans, 2010).

Graph grammars are a popular approach for the generation of action-adventure dungeon levels (see Chapter 5). This is partially due to the effectiveness of the 'mission/space duality' conceptualisation of the generation problem. This is the concept that the structure and order of a player's movement through a playable space is often not intrinsically defined by the structure of the space itself, due to (forced) backtracking, incomplete knowledge, etc (e.g. Fig. 2-1,

from Dormans (2010)). This approach advocates generating the 'mission' first (using graph grammars) and then using that as input to a system that converts the mission representation to a playable space, often also using spatial grammars. This two-phase process naturally lends itself to mixed-initiative also, with opportunity for designer intervention, editing, regeneration etc. of the mission graph before it is reified. This has been an area of significant interest in the literature (e.g. Dormans, 2010, 2017; Lavender and Thompson, 2015; Karavolos, Bouwer and Bidarra, 2015; van der Linden, Lopes and Bidarra, 2013, 2014).

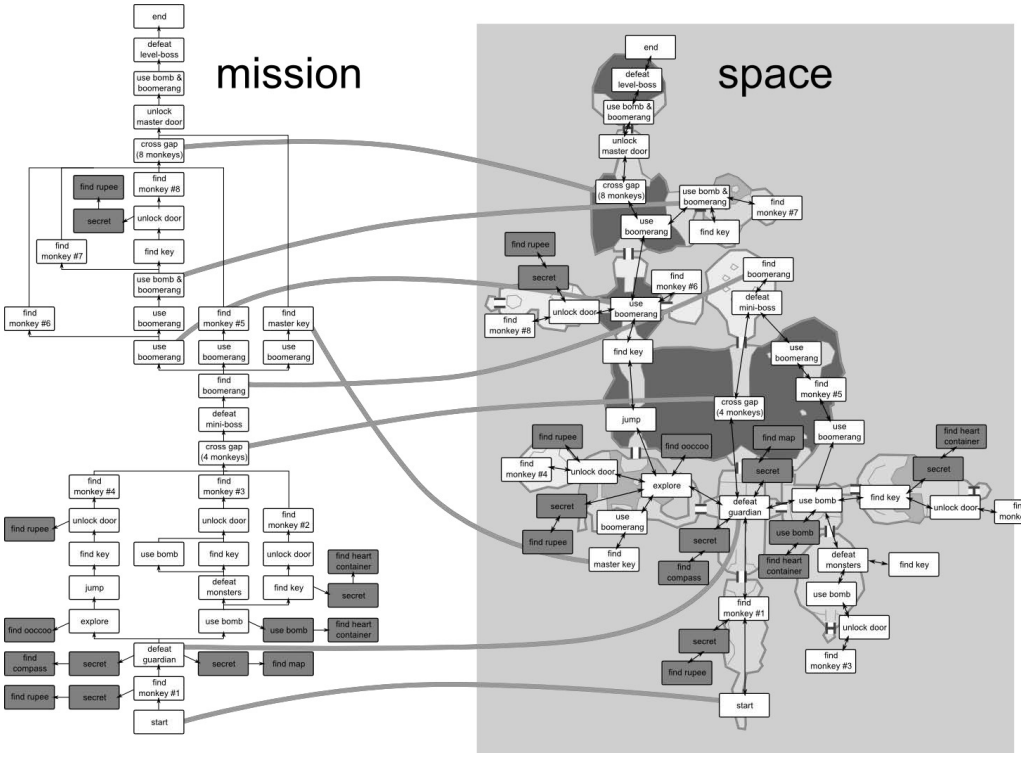



Figure 2-1: Illustration of the concept of mission-space duality as transcribed from the Forest Temple level of *The Legend of Zelda: The Twilight Princess* (reproduced from Dormans, 2010).

### 2.2.2 Search-based PCG

Another popular contemporary approach for PCG are 'search-based' algorithms – so called because they search an implicitly-defined space of possible content using a domain-specific evaluation function to guide successive attempts. Most modern implementations are in the form of 'evolutionary algorithms', though simulated annealing, particle swarm optimisation and stochastic local search are all also represented in literature. The three key elements of an evolutionary algorithm (EA) are a *representation* of individual artefacts to be produced, an *evaluation function* for assessing each individual representation, and a *generation strategy* to define how and when new individuals are produced (Togelius et al., 2011b). The name for the approach arose via analogy to natural selection: the representation can be thought of as the

genotype of a potential individual. The absolute range of all possible valid representations in a given EA is the implicitly-defined search space of content that the algorithm could produce. Typically, EAs maintain a population of individuals, each with an associated *fitness* or *value*, as produced by the evaluation function – which may be an heuristic, such as length or age, or the result of a more complex process such as agent-based evaluation (see Sec. 2.5). Periodically, new individuals will be produced according to the generation strategy: this might involve replacing the least-fit individuals with variant clones of the most-fit individuals, or a more complex approach such as crossover between pairs of successful individuals, or measures designed to maintain diversity within the population. Over time, appropriate search based methods will tend to converge on 'good' or even optimal solutions to problems, where the actual value of 'good' is dependant upon the suitability of the representation chosen and the accuracy of the evaluation function. Given an appropriate formulation of a search space, and a typically domain-dependent technique for converting from representation to actual content, search-based approaches are applicable to a wide range of content production problems, according to Togelius et al. (2011b). This has been another area of significant interest in the literature (e.g. Togelius et al., 2009; Liapis et al., 2013; Yannakakis, Liapis and Alexopoulos, 2014; Yannakakis and Liapis, 2016; Preuss, Liapis and Togelius, 2014; Green et al., 2018; Kartal, Sohre and Guy, 2016).

One of the strengths of many search-based PCG implementations is the ease with which manually produced or altered artefacts can be fed back into the system, facilitating the development of mixed-initiative processes and editors such as Petalz or the Sentient Sketchpad (Risi et al., 2015; Liapis, Yannakakis and Togelius, 2013a). Search approaches have also been applied by Kerssemakers et al. (2012) as a form of evolution to generate appropriately-configured generators.

### 2.2.3 Constraint-based systems

Often, generated content must be 'valid' in some sense — certain properties about it must be true or within particular ranges.Many PCG systems use a 'generate-and-test' technique, typically re-generating any content that does not pass the selected tests. This approach can be wasteful if the generator produces a large percentage of non-valid content, and though it is often possible to adjust the PCG system to mitigate this effect, it will always be necessary to execute the 'test' portion of the cycle, which can be computationally expensive or time-consuming (for more on this, see Sec. 2.5).

Constraint-based systems represent an alternative approach, intended to produce content that is 'correct-by-construction'. Requirements of the finished content are formally expressed as a series of constraints, which are input to a solver able to select assignments of variables that simultaneously satisfy all constraints. In this declarative class of approaches, more focus is placed on 'what' to generate rather than 'how', and though evaluation is still necessary during development to ensure that the generator is producing the expected content, post-processing to test validity of content is unnecessary if validity constraints are properly expressed as part

of the problem definition.

A range of solvers are available to suit differing domains and classes of content. Answer Set Programming (ASP) is a general declarative approach covered in more detail later (Chap. 3), but there are several more domain-specific solvers that are applicable to procedural generation. Tutenel et al. (2009a) describe a custom layout solver developed to ensure that placed objects maintain sufficient clearance in key directions to be usable. Horswill and Foged (2012) use a constraint propagation system for numerical variables relating to the population of a dungeon with enemies and treasure, and Smith, Whitehead and Mateas (2011) uses a third-party library for solving numerical constraints library for procedural platform placement according to rhythm and feasibility restrictions.

Spatial solvers have also been used for placement approaches with size and shape awareness and heuristic-driven backtracking, as by Tutenel et al. (2009a) and Butler et al. (2015).

## 2.3 Mixed-initiative PCG

Rather than replacing the role of human content producers in modern games, interactive PCG systems are increasingly being considered as a tool to augment designers' creativity (Smith, 2014). A number of recent papers (Liapis, Yannakakis and Togelius, 2013a; Smith, Whitehead and Mateas, 2010) consider the concept of 'mixed initiative' generation, where the authorial burden is shared to some degree between the designer and the system.

#### Tanagra

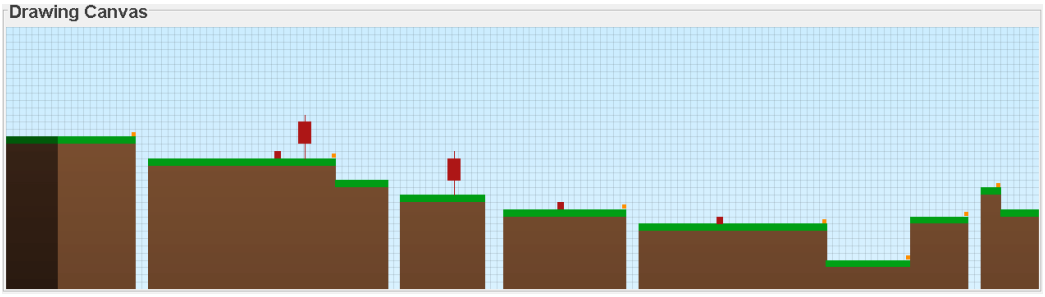



Figure 2-2: Part of a level created by Tanagra and a human designer (from Smith, Whitehead and Mateas, 2010).

Tanagra (Smith, Whitehead and Mateas, 2010) is one such mixed-initiative design tool, intended to aid in the production of rhythm-based platform game levels. A designer is free to add, remove or manipulate platforms and game elements such as springs, enemies and other hazards in the usual way; however a number of additional systems are provided to aid the rapid development of playable levels. At any stage, the human designer may invoke the generator system, which will use the provided game elements as cues from which to construct a complete, playable level without altering or overwriting the designer's work. The system expresses the

geometric relationship between level components as numerical constraints, and uses a constraint solver to ensure that every generated level is playable (Smith, Whitehead and Mateas, 2011). This removes the burden of verifying levels manually from the designer, freeing them to search for alterations that lead to 'fun' levels. The PCG system in Tanagra also provides a 'beat-based' abstraction to simplify the process of manipulating pacing within the generated content. The frequency of necessary player actions is calculated based upon the designer-specified segments, and used to guide further generation. Alternatively, the calculated beat frequency may be directly altered in order to re-generate the level according to the newly provided pace.

#### Sentient Sketchbook

Another mixed-initiative tool, the Sentient Sketchpad system (Liapis, Yannakakis and Togelius, 2013a; Yannakakis, Liapis and Alexopoulos, 2014), provides an interface for co-creatively generating 'map sketches' (low-resolution abstractions) of Real-Time Strategy (RTS) levels. In addition to an editable representation of the current map, the system provides automatic analysis and feedback on six domain-specific metrics, to allow the user to assess at-a-glance salient properties of the work in progress. The key component of the system is the presentation of up to twelve 'suggestions' – minor variations on the current work, produced by a range of genetic algorithms as described in Liapis, Yannakakis and Togelius (2013a) – which may be selected to replace the current map. Six of the suggestions are guided by heuristics that optimise towards each of the domain-specific metrics (such as aesthetics/symmetry and playability/resource distribution), while the remainder are chosen to be as visually diverse as possible while remaining evidently derived from the user's existing design. The system evolves alternate 'suggestion' candidates that the user can swap with their current sketch at any time, or simple use as a visualisation of what a 'more symmetric' or 'more balanced' map could look like. The intention is to provide a system that allows effortless exploration of a local space of maps inspired by an initial outline: once a sufficiently satisfactory sketch has been developed/located, an automated process converts it into an equivalent higher-fidelity RTS map. This presents a potential weakness of the system: the amount of computation necessary to calculate updated values for the metrics and produce all necessary suggestions is only feasible for highly abstract representations; in the implementation described these are 8x8 maps (64 tiles). The authors propose that the constrained size aids designer comprehension of the sketches, and demonstrate translation of the output to a range of concrete forms, including both strategy maps and rogue-like dungeons. They suggest that with creative re-interpretation of the metrics and representation, other formats such as FPS levels should also be possible.

#### Ludocore

Ludocore is a concrete implementation of the mission-space duality concept: graph-grammar generation of a level 'mission' (schedule of player actions), reified by spatial grammar into playable space that supports that mission. Karavolos, Bouwer and Bidarra (2015) describe a version of the system that supports mixed-initiative production of game levels; van Rozen and

Heijn (2018) describes further work on visualising the effects of changes to graph grammars, and Lavender and Thompson (2015) apply the approach to concepts from an existing game (Zelda). The cyclic generation refinement led directly to the creation of the game Unexplored (Ludomotion, 22 February 2017), and recently to Unexplored 2.

In Karavolos, Bouwer and Bidarra (2015) a multi-stage generation system is developed, which performs a number of iterations of successive refinements upon increasingly detailed and complete representations of the content. Initial productions are based upon a series of generative grammar systems, while later refinements involve selection of appropriate chunks from a library of pre-authored content. Although the system is capable of running without designer intervention, at the end of each stage the user is given an opportunity either to approve the changes, modify the current representation manually, or instruct the system to re-generate the previous stage (resulting in different output). This affords a degree of direct designer control over the final output of the system, without requiring that the designer manually perform each individual stage themselves. As with Sentient Sketchpad, automation of individual steps frees the designer to explore quickly and easily the space of alternative representations for the desired content, and the authors show that the system may be generalised to produce either platform game levels or action/adventure style dungeons. However, the system is highly reliant upon suitable, domain-specific grammars and pre-authored content libraries, and the validity of generated content depends on the correctness of multiple grammars and the presence of appropriate features in the pre-authored content chunks.

### 2.3.1 Player-driven mixed-initiative PCG

Increasingly, the performance of PCG systems is sufficient for use at runtime, and therefore able to incorporate information about the player's actions into their decisions. In some cases, the player is actively and deliberately encouraged to participate in this interaction, and so the creation of the world (or other content) becomes a joint activity between the player and the system.

#### Charbitat

In Charbitat (Nitsche et al., 2006), individual sections of the world are generated to reflect one of the five elements of Taoism. Which element is chosen, and the strength of the affinity, is dependent both on the qualities of nearby sections and on the player's actions within the game world. Certain elements of the game world will only appear in sections with particular properties, and so the player must indirectly guide the generator towards the production of appropriate sections via their own in-game behaviour.

#### Anza Island

In comparison, player agency in Anza Island (Compton, Smith and Mateas, 2012) is more explicit, although still indirect. The PCG system in this case is responsible for the configuration of pathways connecting pre-authored elements of the game world, and deliberately attempts to

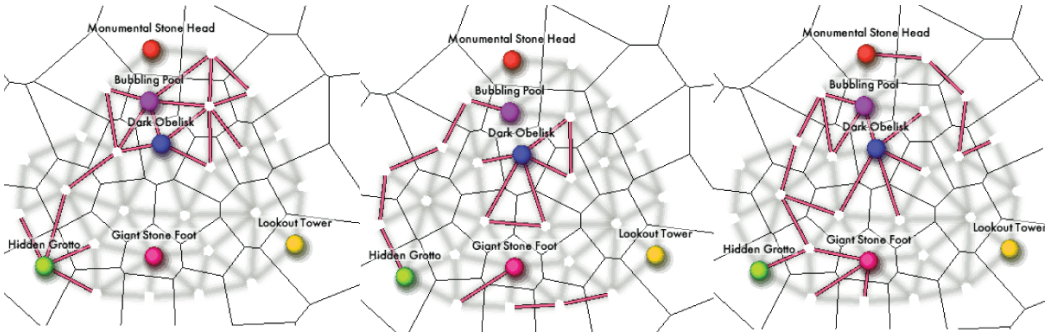



Figure 2-3: Possible layouts chosen by the Anza Island system (from Compton, Smith and Mateas, 2012).

select configurations that make it difficult or impossible for the player to achieve their goals – as in Fig. 2-3, where many of the landmarks are not connected and so the player cannot reach them. The player's objective is to travel throughout the game world and visit each landmark, and they are equipped with the ability to place a number of specific restrictions upon the PCG system to ensure favourable configuration of pathways, with additional different restrictions that can be unlocked throughout gameplay. In this way, the player may indirectly influence the (re-)generation of the game world, in service of their ultimate goals.

## 2.4 Industry PCG

In this section we present a selection of interesting industry approaches to PCG. These are frequently comparatively simple construction/selection approaches — the two missing categories from Smith (2014). Several aspects of typical PCG systems are key considerations for commercial application: (i) PCG can produce artefacts that are unpredictable/unlearnable/unguideable (Smith, 2014) (ii) successful game contexts give the player agency to fix things if the generator produces 'unplayable' content: either by providing diegetic tools to remove obstacles (*Spelunky*, *No Man's Sky*, *Minecraft*, *Terraria*), or allowing the player to easily abandon bad content in favour of seeking a better seed elsewhere (*Minecraft*, *No Man's Sky*, *Borderlands*) (iii) PCG is a gateway to User-Generated Content (UGC): *Spore*, *Galactic Arms Race*, *Petalz* (and also Tracery (Compton, Filstrup and Mateas, 2014), 'casual creators' (Compton and Mateas, 2015): not exactly industry, but definitely non-academic) (iv) there is philosophical distinction between stochastic and deterministic PCG: the former provides variety, the latter is more akin to compression but can be psuedo-randomly seeded.

PCG was historically a natural design response to limited storage space; early games such as *Elite*, *Rogue* or *Frontier* used it as a force multiplier for content creation, and this approach lives on in their successors.

The 'endless runner' genre of mobile and web games predominantly use simple PCG for variation — *Flappy Bird*, *Robot Unicorn Attack* and *Temple Run* encourage players to develop quick judgement and responses rather than memorising layouts. These are PCG by construction —

large ('endless') environments produced through simple linear combinations of a comparatively limited pool of kit parts.

Some ('indie') games such as *Ultima Ratio Regnum* (*URR*) or *Lenna's Inception* use PCG as a vehicle for unique player exploration — through generation of vast but finite and interesting worlds, alongside a known toolkit of abilities that facilitate navigating and discovering those worlds. *URR* produces a world containing deep impactful histories and populated with the cultures that have have taken part in and been shaped by those histories. *Lenna's Inception* is a novel generated action-adventure game in the long tradition of the *Legend of Zelda* randomisers, that produces a complex overworld map containing structured access to a series of dungeon levels that contribute to and rely on increasing player character capabilities.

*Galactic Arms Race* (*GAR*) and *Petalz* employ PCG as a directable aid for UGC — both games include systems for the generation of content where the survival/assessment heuristic is explicit player choice. In *GAR*, the cgNEAT genetic algorithm is responsible for the diverse properties and behaviours of available player weapons (Hastings, Guha and Stanley, 2009). 'Successful' weapons are the ones that players explicitly choose to spend time using, presumably for power/aesthetic reasons, and so gain a larger share of the population. In *Petalz* (Risi et al., 2015), a similar concept is used to provide varied flower designs that players may plant in their virtual gardens or share with friends — in both cases, part of the gameplay consists of exploring and making decisions about artefacts generated through PCG (as in Cook et al., 2016).

Several games (*No Man's Sky*, *Elite: Dangerous*) use aspects of 'multi-level, multi-content' PCG to generate solar systems, planets, and their contents. Outer space exploration environments are a popular theme amongst games that contain procedurally generated content, as they provide two forms of 'insulation' against the effects of low-quality outputs from a generative system. The first is psychological — players are already primed for oddness and unrecognisable geology or organisms. The second is freedom of scale — when presented with content that is unappealing for whatever reason, the player has the freedom to simply leave and seek some other part of the galaxy, where things will be different (Compton and Mateas, 2017, describe this 'search' process in more detail). A benefit of deterministic generation according to stellar location is that social sharing of the coordinates of particularly useful or attractive in-game weapons, vehicle, creatures or planets is possible and rewarding. This is a mechanically-driven subset of the phenomenon of social sharing and appreciation of 'interesting' outcomes from generative systems

An impact of the decision to use PCG in games is frequently a low-resolution representation of the world (i.e. tiles, voxels) compared to other modern games. Frequently this goes hand-in-hand with increased player agency in terms of being able to reshape the world around them. From a game performance point of view this is computationally simplified by the low-resolution representation, whilst from a game design perspective it can provide a useful alternative for situations where the generator produces content that would otherwise be impassable, such as large cliffs or obstacles blocking doorways. In turn, this allows for riskier generation practices, that increase the chance for emergence of desirable novel configurations whilst also making undesirable (but now still useable) configurations likelier as tradeoff.

The indie permadeath Role-Playing Game (RPG)s *Unexplored* and *Unexplored 2: The Wayfarer's Legacy* make use of a specific form of graph grammar to produce their overworld and dungeon maps, based on work by Dormans (2017). This is an application of years of academic research to build a commercial indie game using grammar-based cyclic generation techniques — i.e. graph grammars where all operations add loops or preserve existing ones in new forms, in order to facilitate generation of more complex level structures including short-cuts and one-way connections.

Named for a conceptually similar phenomenon in quantum physics, the Wave Function Collapse (WFC) algorithm[2] allows for generation of near-endless variations on a common theme, by deriving connectivity information for a series of kit-parts from a set of designer-provided exemplars. This has been very popular amongst both indie developers (*Bad North*, *Townscaper*, *Caves of Qud*) and academic researchers (Karth and Smith, 2017) as the algorithm is fairly simple to implement whilst still producing impressive results that typically don't contain the occasional procedural weirdness that PCG is otherwise often known for.

An early application of ASP to the production of game content by Schanda and Brain (2009) specified rules and facts relating to base layout in the open source RTS *Warzone 2100* project, with initial support for generating entire game maps according to global and local constraints (Smith, 2012).

## 2.5 PCG evaluation methods

As the field of PCG research matures, increasing attention is being paid to the development of appropriate methods for evaluating the output of PCG systems, in order to ensure that they are functioning as intended and also to facilitate comparison between varying approaches. Given the typically large output spaces of most techniques, human qualitative evaluation of more than a few specific concrete artefacts is difficult, and even quantitative analysis is often dependent on taking a constrained sample of the output population. Since many procedural generators combine multiple systems, it is important to attempt to ensure that unexpected interactions will not lead to content that is not suited for its intended purpose, or even unplayable. Therefore, a number of automated evaluation methods have been proposed, either to verify during design and development that the generator is producing expected output, or to validate at runtime that a particular generated artefact is fit for purpose. The individual requirements of a particular use-case may also necessarily influence the choice of generation approach used — elements that are decorative or not critical to player progress (music, character appearance) are not required to provide the same guarantees of appropriateness as, for example, level geometry or quest structure, where mistakes in generation may leave the game in an unplayable state.

Visual evaluation of sample outputs generated during the development of a PCG system may indicate basic inaccuracies, however for generators with large output spaces this is often insufficient to guarantee correct output in all cases. For many procedural generation approaches it would be difficult to guarantee correctness formally, however a range of evaluation techniques

[2] `https://marian42.itch.io/wfc` — Accessed 13/12/2022

have been developed to analyse individual outputs, samples of the output space or even constituent elements of the system itself, in order to help ensure that generated content will be fit for purpose (Shaker, Smith and Yannakakis, 2016). Completion feasibility (or other related properties) of content can be verified at point-of-use using agents designed to consume the content in a manner similar to players – if the agent can complete the level, the player should also be able to. Metric-based computational analysis of (a sufficiently large sample of) the output space can also serve to indicate biases or omissions in the space of content the generator is capable of producing, in order to inform further development or corrections – the identification of suitable metrics for this approach is an active research area. In certain contexts, it may also be feasible to include human analysis of content elements as part of the generation process: either as a pre-processing step or as an incidental part of gameplay (systems that include this process as core gameplay might more accurately be described as user-generated content, which is beyond the scope of this report). Several of these possible evaluation approaches are described in more detail below.

### 2.5.1 Agent-based analysis

One approach to assessing generated content automatically is the use of simulation-based evaluation: that is, an agent is developed which is able to 'play' the generated content in some sense, and the results of its attempts to do so are used to evaluate the content. Togelius, Justinussen and Hartzen (2012) describe the use of two simple A* agents to evaluate dungeons produced by an evolution-guided ASP program (see Sec. 3.3.4). One agent follows an additional heuristic causing it more closely to mimic the behaviour of a skilful player, and therefore the difference in performance between the two agents represents the degree of 'skill differentiation' provided by a particular generated dungeon. They propose that a degree of player influence over the outcome of attempting a particular dungeon is a desirable property of their generated content, and suggest this paired-agent approach as a suitable technique for assessing this property.

One weakness of this approach is the additional burden of developing an agent able to accurately mimic a player's experience of the generated content. To ensure the connectivity of rogue-like dungeons, simple A* search or flood-fill algorithms may be sufficient, however other domains require more complex approaches. There is a growing area of research looking at the development of 'General Video Game AI' (GVG-AI) agents that are able to attempt to play any given game or content, with the intention that they would be appropriate for this task (Perez-Liebana et al., 2019).

For platform level generation, unplayable content may easily be produced if obstacles are too high or gaps are too wide for the player to progress past them. The process of detecting these generation failures can be complicated by the presence or absence of other platforms, enemies or power-ups (temporary boosts to player abilities) — of validity as a consequence of the generation process, it may be necessary to test automatically generated content for the existence of a feasible route to completion. If a suitable agent with access to the same affordances as a player is able to complete the level, this indicates that a feasible route exists.

Other properties of the level, such as expected damage taken, or expected points scored, may also be estimated from the agent's performance (Horswill and Foged, 2012). Sufficiently simple agents execute quickly enough to assess content in this manner at a rate that makes this method suitable for live generation of level. The development of platform-game-specific agents for this purpose may be seen as a precursor to the work on GVG-AI[3] , and some of the techniques since developed for that field are now being applied to platform-game-playing agents in turn (Togelius et al., 2013b; Jacobsen, Greve and Togelius, 2014).

### 2.5.2 Domain-general metrics

Another approach to automated assessment of generated content is the calculation and analysis of various metrics across representative samples of a generator's output. Each metric provides an easy way to assess specific properties of the output, and different metrics are applicable to different classes of content. Domain-general metrics would support an independent package of analysis tools that could provide basic evaluation of any PCG systems. Domain-specific metrics, while less widely applicable, provide more detailed insight into relevant aspects of a generator's output, and facilitate comparison between different generators for the same content type, or even between different parameter values for the same generator. Suitable domain-specific metrics are also more easily identifiable for given content.

This topic is also covered in detail in Chapter 6. In this chapter we focus on the overview, related literature and history and development of the approach, and in Chap. 6 we provide a more technical view of the process, focusing how we have applied the steps described by Smith and Whitehead (2010) and comparing our results to another example within our specific domain.

Smith and Whitehead (2010) suggest a four-step approach to analysing the expressive range of a PCG system: (i) determine appropriate metrics, (ii) generate content, (iii) visualise output space, and (iv) analyse the impact of parameters. In this approach, the metrics chosen should lead to informative visualisations that can aid the system designer in identifying irregularities in the output that may be a consequence of specific design decisions or poorly-tuned parameters. As a demonstration of this approach, the authors apply the metrics of '*linearity*' (summed deviance from a linear regression performed across platform heights) and '*leniency*' (weighted sum of challenge items present, approximation for difficulty) to a range of platform game levels generated by the Launchpad system (a precursor to Tanagra (Smith, Whitehead and Mateas, 2011)). 10,000 levels are sampled and the results plotted on a 2D histogram with the axes defined by the range of metrics scores. Variance of the output space is demonstrated by repeat evaluations with different parameter settings, and the effect on the output space of a particular implementation detail is noted. Though the metrics provide an initial method for generator output space analysis, the authors note that they are both domain-specific (and therefore not usable for other content kinds) and insufficient to capture many aspects of generator output variance.

[3] `https://sites.google.com/site/platformersai/` — accessed 27 November 2020

This approach is extended by Shaker et al. (2012), who suggest two additional metrics: *density* (the sum of occurrences of alternative paths at any given point), and *compression distance* (a pairwise comparison metric, the sum of differences between compressed forms of two levels from a generator, averaged across all pairs from that generator). Normalisation of all metrics is also performed, in order to facilitate comparison between generators. They note that this approach could be used by designers to test and compare different generators within the same game genre, or by developers to highlight limitations in the expressivity range of generators – and they suggest potential solutions to some of the limitations observed in the tested generators.

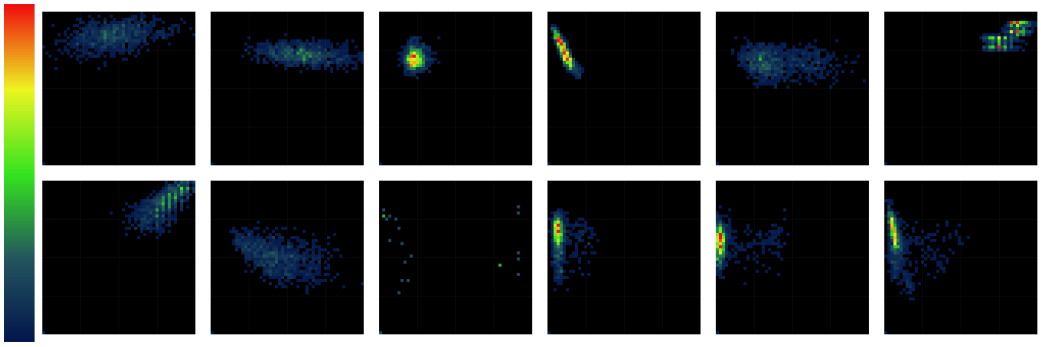


Figure 2-4: Density-Leniency comparison across six generators & variants (Horn et al., 2014).

A later work in the same area by Horn et al. (2014) makes use of the four-step approach, the four previously listed metrics, plus *pattern density* and *pattern variance* to compare output visualisations for 1,000 samples each from seven platform level generators, and 22 original levels from *Super Mario Bros 1.* An example result from their approach is shown in Fig. 2-4 — each of the 1,000 samples is plotted on a heat map where the y-axis represents *density* and the x-axis is the *leniency* metric (both normalised). Considerable variance is shown between the output spaces of different generators, and even between different parameter settings for the same generator. In some cases, distinct clustering or other artefacts are observed, indicating that certain generators might be unsuitable choices for particular applications. None of the surveyed generators exhibit an output space significantly similar to the original levels. The authors suggest that the comparison provides a baseline by which to characterise other platform level generation systems, and make available code and samples to facilitate future comparisons.

In order to develop further the concept and investigate the application of these metrics to hand-designed levels, Canossa and Smith (2015) performed a user study on platform game level design and analysis with student designers (sample size 28). Initial characterisation of the six listed metrics over levels targeted at four specific 'moods' revealed insufficient differentiation to identify mood via computational analysis. This led to the development of twenty new possible metrics, each with a suggested method for quantitative analysis, although it is noted that many of these would currently be infeasible to assess computationally — either because they relate to aesthetic considerations (sound design, visual appearance) or psychological impact of the design on the player (threat level, reasonability). Finally, the authors investigate the domain-generality

of the new metrics by studying their applicability to a first-person action/adventure puzzle game *Portal 2* (Valve Software, 18 April 2011)[4] , and note that eighteen of the metrics are 'portable' (domain-independent for these examples) and therefore applicable to both platform-game and action-adventure genres.

Despite the wealth of research in this area, initial empirical evaluations of these general evaluation metrics for platform games indicate that computational analysis does not always agree with the results of actual user studies, and therefore while it may be appropriate for preliminary exploratory study, it should not replace traditional user-driven evaluations. A recent paper by Marino, Reis and Lelis (2015) compares computational analysis of levels produced by a range of generators for the Infinite Mario platform game research platform (Togelius et al., 2009; Reis, Lelis and Gal, 2015; Mawhorter and Mateas, 2010) with the outcome of a user study (n.=37), and found little correlation between player responses and the qualities suggested by metric analysis (*leniency* and *linearity* (Smith and Whitehead, 2010), *density* and *compression distance* (Shaker et al., 2012)). Although Marino, Reis and Lelis (2015) suggest a number of specific weaknesses with certain of the metrics, they also note that the user study analysed factors such as 'enjoyment' that the computational measures specifically did not attempt to address. Additionally, due to concerns about participant fatigue and practicality, each user was only presented with one level from each generator, which suggests possible issues with sampling that could affect the strength of the conclusions reached, which were specifically that (i) computational metrics are inaccurate for estimating enjoyment, (ii) limited with regards to visual aesthetics, and (iii) misleading about perceived difficulty.

Cook, Gow and Colton (2016) demonstrate application of this approach in a tool for the game engine Unity, '*Danesh*' that supports live re-generation of the metric visualisations, and automated exploration of generator parameters to search for desirable distributions. The ability to investigate the outputs of different parameter configurations simplifies the gulf of evaluation when assessing the behaviour of a generator

Sampling a large number of generated examples can help to provide an intuition for the characteristics of the generator's typical outputs, and can assist in inspecting the effects of tweaks and changes. However, this method of analysis alone cannot guarantee the absence or impossibility of specific malformed outputs, if examples of those deficiencies happen to not be among the outputs sampled, or are not adequately reflected in the chosen metrics. A number of other evaluation approaches are described below.

### 2.5.3 Other approaches

In contrast to agent-based or statistical automatic analysis of content, some PCG systems still rely on human evaluation in which use is made of variations on crowd-sourcing techniques to perform evaluation at a scale that is able to compensate for the potential scope of the generated space. There are also a number of techniques used to deal with the problem of potentially unusable content, which vary with the nature of the system.

[4] `http://www.thinkwithportals.com` — accessed 27 November 2020

Reis, Lelis and Gal (2015) make use of human computation explicitly via a paid crowd-sourcing platform. Participants are paid to evaluate small procedurally-generated chunks of platform game levels, tagging individual chunks with useful (but necessarily subjective) information about difficulty, aesthetic appeal and enjoyment, and rejecting them outright if they are invalid in some sense. Taken together, these chunks represent an annotated library of pre-authored content, which the system is able to assemble into complete levels based upon the provided metadata. This method of evaluation helps to ensure that players are never exposed to invalid or undesirable content, as it will have been filtered out by paid system participants performing explicit human evaluation.

In contrast, Hastings, Guha and Stanley (2009) rely upon implicit human evaluation and easy availability of valid alternative content to sidestep possible problems caused by invalid generated content in their game *Galactic Arms Race*[5] (*GAR*). Weapons in *GAR* are represented as particle systems produced by the cgNEAT algorithm for evolved neural networks, where the fitness function is assessed via implicit human evaluation. The genetic algorithm used considers a population composed of all of the weapons currently equipped globally by *GAR* players, and produces new weapons based upon that population. As alternative weapons are always easily available, the system assumes that players will not continue to use broken (i.e. invalid) weapons, and relies on 'the wisdom of the crowd' in order to gradually improve the general fitness of the weapon population over time.

## 2.6 Discussion

As described in the preceding sections there are a range of existing approaches to PCG represented in both literature and commercially published games. Increasingly the use of PCG represents an authorial choice that can land anywhere between full construction, co-creation and dynamic runtime generation. The integration of PCG with games requires careful assessment and design choices relating to intent and affordances; there are still many open questions about how best to represent and generate artefacts for many kinds of context.

In Togelius et al. (2013a) a collection of prominent PCG researchers set out an overview of the three 'grand goals' of PCG research as informed by and extrapolated from existing research directions in literature, namely '*Multi-level Multi-content PCG*', '*PCG-based Game Design*' and '*Generating Complete Games*'. They also detail eight research challenges that they see as achievable topics in contemporary PCG, including the requirement for '*General Content Generators*'. As they define it, 'A general content generator would be able to generate multiple types of content for multiple games. The specific demands, in terms of aesthetics and/or in-game functionality, of the content should be specified as parameters to the generator.' — Togelius et al. (2013a), Sec 3.3. This work attempts to address that challenge, and by doing so provide progress towards the *grand goal* of 'Multi-level Multi-content PCG'.

[5] `http://galacticarmsrace.blogspot.co.uk/p/research.html` – accessed 28 November 2020

# Chapter 3

# Answer Set Programming

Answer Set Programming (ASP) is a declarative logic programming approach aimed at modelling constrained combinatorial search problems. ASP problems are specified by asserting appropriate facts, rules and constraints relating to the domain of interest, and a domain-independent solver is able to return all sets of supported mutually-comprehensible facts that satisfy the assertions.

> "We claim that it is easier to sculpt a design space by repeatedly carving away undesirable regions (identified by describable flaws) than it is to guess a procedure which implicitly defines the same space."
>
> *Answer Set Programming for Procedural Content Generation: A Design Space Approach* (Smith and Mateas, 2011)

ASP is an approach that has recently shown significant promise for Procedural Content Generation (PCG), by using a domain-independent off-the-shelf logic solver to select valid outputs from a constraint-bound space of possible solutions (Smith and Mateas, 2011; Brewka, Eiter and Truszczyński, 2011). This allows the design space to be modelled declaratively in terms of facts, possibilities and conflicts, and then some or all of the possible valid answer sets may be generated efficiently by the solver without requiring domain-specific search algorithms. These answers each represent only instances of the content that satisfy the specified hard gameplay constraints in the provided formulation, which may include concepts such as required connectivity between areas, or solution existence and uniqueness in the case of puzzles (Smith, Butler and Popović, 2013).

ASP is a declarative programming approach, where knowledge about the domain of interest is formally represented as a series of facts and constraints (for a more thorough background on ASP, see Sec. 3.1). This power comes at the cost of applying a range of domain-specific techniques to limit the potential combinatorial explosion in the answer set space (Smith and Bryson, 2014).

Modern ASP solvers are highly optimised, and able to select one or all valid answer sets from a given problem description very quickly. However, despite a range of techniques developed to

learn and prune entire infeasible subspaces of potential answers, a badly designed answer set program can still lead to combinatorial explosion of processing needed in the worst case, and so care must be taken to ensure that domain and problem encodings are designed to allow efficient solving.

In the remainder of this section, first we discuss a range of introductory materials for ASP and related concepts, followed by a survey of existing applications of ASP to generating content for games and puzzles, or other relevant content. Finally, we consider a number of approaches used for simplifying the authoring process for ASP problems.

ASP is becoming an increasingly popular tool for solving constrained problems, with a highly-recommended and thorough introduction to the approach presented by Brewka, Eiter and Truszczyński (2011) in a widely read general-interest computing magazine (Communications of the ACM). However, the field of ASP research is still comparatively recent, and there is not yet comprehensive support for easy development and debugging, as noted by Brain and De Vos (2008) and Boenn et al. (2011). A number of existing projects attempt to avoid this issue by implementing alternative authoring approaches, as covered in Sec. 3.3.4 below. Alternatively in lieu of user-friendly debugging and editing tools, Brain, Cliffe and De Vos (2009) present a number of recommendations for structuring efficient approaches via incremental tested development, and describe a methodology suitable for encoding problem solving in ASP in an intuitive manner.

## 3.1 Answer Set Programming (ASP) fundamentals

AnsProlog is a text-based declarative programming language that can be used to concretely specify knowledge, rules and constraints relating to a domain, rather than the ordered sets of properties and procedures specified by imperative languages. This allows the design space to be modelled declaratively in terms of facts, possibilities and conflicts, and then some or all of the possible valid answer sets may be generated efficiently by the solver without requiring domain-specific search algorithms.

As covered in Chap. 2, various techniques have been employed to generate content for games (including grammar-based productions, genetic algorithms, machine learning and agent-based assembly). Amongst these is the use of constraint-driven approaches, including ASP. ASP is attractive for certain forms of structured content generation for several reasons: 1. It is non-monotonic, meaning that the introduction of new facts may result in the retraction of previous inferences. This makes it suitable for the generation of tightly constrained content, and in particular mixed-initiative generation where portions of the content are specified explicitly by human designers (see Sec. 2.3) As further portions of the content are specified with certainty, contradictory possibilities may be eliminated. 2. ASP operates under the 'open world assumption', where facts unspecified are unknown and open possibilities (in contrast to the 'closed world assumption', where unspecified facts are explicitly treated as false). This can simplify some generation approaches by allowing certain decisions to be deferred. 3. Fully declarative — the order in which rules and facts are declared has no bearing on the final outcome, and this

makes it easier to compose a problem formulation out of several single-purpose fragments. 4. Content- and algorithm-agnostic. ASP operates on the level of abstract specification of facts. Though this means that a translation layer is required to convert the produced answer set(s) into playable content, it also enforces a separation that means changes can easily be made to the problem specification without requiring code rewrites in other areas.

ASP program formulations are Prolog-like declarative code files consisting of facts and rules, which may be passed to one of a number of 'off-the-shelf' solvers[1,2] for reasoning over; one of the advantages of this approach is that the hard algorithmic work can be performed by a highly-optimised library developed elsewhere and can benefit from future updates. We use Clingo 5.2.0 with Python as an external scripting language for output processing.

> "The hardest and most central problem in AI is, and may always be, this:
>
> Expressing, to a machine, the world we want."
>
> Kate Compton, Twitter (2016)

Briefly: Answer Set programs consist of a collection of declarations, which may be considered facts, rules or constraints. Rules produce a 'head' fact, the truth of which is implied by the collection of terms that form the body of the rule. Facts are a special case of rules that have an empty body and so the head is automatically true. Constraints are a special case where the head is empty; answer sets where the body collectively evaluates to true are discarded. Terms may be constants, variables, or functions that combine variables, constants and/or other functions.

You specify a problem by encoding relevant information about the domain in terms of facts and rules in AnsProlog syntax, which may be passed to an off-the-shelf solver. Solvers are optimised to efficiently select and/or enumerate valid answer sets for the specified problem, by 'grounding' out each rule over all valid variable values, and then searching for assignments that satisfy all constraints.

### 3.1.1 Syntax

Full AnsProlog syntax and definitions are discussed by Brewka, Eiter and Truszczyński (2011). AnsProlog problem encodings take the form of multiple declarative rules.

```
reachable(Area) :- door_connecting(start,Door,Area), not locked(Door).
```

Facts are a rule with no body, where by implication the head is true.

Rules consist of a collection of literals of the form L0 ← L1, ..., Lm, not Lm+1, ..., not Ln. They are declarative statements about the domain of interest, and may be read as "**if** every term in the body is true **then** the term(s) specified by the head is(/are) true" or "the truth of H is *implied* by the conjunctive truth of L1–Ln".

[1] Clingo: `https://potassco.org/clingo` — accessed 27 November 2020
[2] `dlvhex`: `http://www.kr.tuwien.ac.at/research/systems/dlvhex` — accessed 27 November 2020

Terms may be constants, variables, or functions that combine variables, constants and/or other functions. They are 'ground' if they contain no variables; otherwise must be later expanded during grounding to represent each value for each contained variable.

Variables are terms that may take on any of a range of constant values. In AnsProlog syntax they are represented by names that begin with an uppercase letter.

Constants are atomic identifiers for items within the domain of interest - in AnsProlog syntax they are represented by names that begin with a lowercase letter or digit.

```
cup(1..3).
1{ hidden(C, pea) :cup(C) } 1:- pea.
```

A particularly powerful part of the ASP syntax is the specification of disjunctive heads, or 'choice rules', where the number of head facts produced by a given rule can be constrained to within certain bounds, across all possible values. In the example above, the '`X{...:...}Y`' syntax specifies that at least `X` and at most `Y` facts should be deduced; in this case precisely one `hidden/2` fact will be present per answer set.

ASP provides for two kinds of negation: "classical" negation ($\neg a$) meaning that it is known that $a$ is not true, and negation-by-failure (**not** $a$) meaning that it is not known whether $a$ is true.

Constraints are specified as rules with an empty head: intuitively, if the existing assignments of literals is such that the empty head is implied `true` by the rule's body then some previous assignment is invalid and this answer set should not be emitted.

### 3.1.2 Grounding

Any rule or term containing only constants and functions of constants is considered 'ground', while any rule or term containing reference to one or more variables is 'nonground'. In order to solve for the answer sets of a problem encoding, all non-ground rules must be replaced with all possible ground versions across all values each variable could take. In some cases this can lead to ground representations of a program that are far, far larger than intended, due to poor choice of abstraction for the encoding.

Sec. 3.2.1 describes one approach for decreasing the size of ground problems: decomposing the problem into two or more feed-forward steps. This can greatly reduce the scope of each sub-problem, but is not feasible where high-level decisions are dependent on 'later' low-level decisions, unless a mechanism for pre-generating and adhering to the decided low-level outcomes is developed. Another approach for reducing the ground size of the problem is to optionally externally determine and then specify the value of some variable as a passed parameter constant.

> "ASP: It's basically creating a multitude of possible universes, and then CULLING THOSE THAT ARE NOT TO YOUR LIKING."
>
> Stella Mazeika, Twitter (2015)

## 3.2 Working with AnsProlog

In this section we cover a number of the practical aspects of working with AnsProlog: the choice of solver, the decomposition of a problem into multiple files representing different aspects, feeding generated facts back in to the process, parameters (rand_freq, seed), 'hosting' of ASP approaches via python, C++ libs and emscripten.

A range of 'off-the-shelf' solvers are available for working with ASP. In this thesis we document experience with Clingo and `dlvhex` (see Sec. 4.5.1), however DLV and others exist; in some cases built on previous SAT solvers. Different projects varying subsets of the ASP language standard and additional features like external atoms.

As described in Sec. 3.2.1 it is standard practice to divide a problem representation into loosely self-contained modules: data representation, initial fact inference, expansion rules, constraints and their supplemental facts; etc. These modules can then be ground and solved together, or potentially used sequentially in order to reduce the size of the ground representation. The output format from an answer set solver is one or more sets of self-consistent facts; with minimal processing these facts can be used as input to another stage of the process.

### 3.2.1 Output processing

A common approach to reducing the complexity of procedural 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 (2011) describes a method using graph-rewriting systems to generate an abstract mission-model of the desired gameplay within a level prior to translation to a usable dungeon play space; Karavolos, Bouwer and Bidarra (2015) provide an extension supporting mixed-initiative translation to two different game genres; and Lavender (2016) applies generative grammars to use this technique for tile-based action-adventure levels. Smith et al. (2012) make use of a similar approach using ASP to generate high-level puzzle descriptions and then produce valid playable instantiations of those puzzles.

One challenge associated with using ASP is appropriate formulation of the problem in order to minimise the size of the ground representation of the problem and hence the time taken to ground and solve. One option for minimising total problem size for some applications is to split the problem into suitable subtasks, which can be ground and solved individually, akin to typical feed-forward generation. The basic implementation of this refinement approach involves selecting suitable atoms from the output of an initial ASP run and loading them as part of the input for a successive child problem — this can help to ensure that each program is only grounding and solving over variables relevant to the subtask at hand, at the cost of losing the ability for inferences derived during the current program to affect the generation of answer sets in the parent. This is similar to the multi-stage ASP used in *Infinite Refraction* (Butler et al., 2013) and proposed for dungeon elaboration (Smith and Bryson, 2014), and differs from the 'iterative solving' functionality offered by Clingo.

In *Infinite Refraction*, Butler et al. (2013) make use of ASP to generate abstract puzzle

descriptions (in terms of required concepts) as part of the pre-generation for an overall progression. Individual puzzle requirement specifications are then refined into complete puzzle specifications by a separate ASP processing step. One possible approach is to integrate a semantic model of the 'narrative' to guide successive, iterative generation steps; using a range of smaller ASP programs that gradually refine an abstract model of the generated content into the final output.

### 3.2.2 Domain specific dialects

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.

For small problems, in-program specification of ASP facts and constants is manageable and simple, but for large or complex domains it appears advantageous to provide a mechanism for automatic provision of these data, whether generated algorithmically, or translated from some external source. With a formally defined desired design space in some suitable description language, it should be possible to automatically author an ASP representation of that space, and use this to generate or verify content within that space.

Novel Domain-Specific Language (DSL)s such as Inst$AL$ (Cliffe, 2007) 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 (Neufeld, Mostaghim and Perez-Liebana, 2015). Finally, Gaggl, Schweizer and Rudolph (2015) present a mechanism for translating the Web Ontology Language (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.

Beyond simply searching for appropriate parameters, a number of automated approaches have been investigated for simplifying the process of authoring complex ASP problem encodings. Cliffe (2007) describes a domain specific language (Inst$AL$) for reasoning about institutions, that may be automatically translated to an ASP representation of the given scenario in order to simplify the process of formal institutional modelling. To support the use of ASP for reasoning about institutions, Cliffe (2007) provide a translation process from a DSL, Inst$AL$, to an ASP model. Inst$AL$ is an 'action language' designed to model the progression of events in a domain over time, and as such is semantically closer to the methods used to model institutions. It is designed to be human-readable and succinct, and can be automatically translated into a formally-equivalent ASP model, in order to reduce the burden of authorship and eliminate the potential for programming errors. The Inst$AL$ syntax and file format are both highly constrained, and tailored to nature of the institutional modelling problems being considered. This simplifies the process of verifying and translating the specifications, at the cost of restricting the capabilities of the language.

## 3.3 Answer Set Programming in games

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 - explicitly specifying a space of possible artefacts to be generated, and thus allowing a solver to select valid instances from within this space. Early uses include generation of strategy game maps and both educational puzzles and puzzle progressions (Smith and Mateas, 2011; Butler et al., 2013).

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 (Neufeld, Mostaghim and Perez-Liebana, 2015).

Often hard constraints are important for gameplay or implementation reasons — these could include ensuring that keys are available before the doors they unlock, or restricting possibilities for players to trap themselves in infinite loops. As the network of constraints on a given problem becomes more complex, the problem becomes less suitable for search-based approaches as valid solutions may no longer be meaningfully 'adjacent' within the representation space and significant time may be wasted producing candidates that do not pass the increasingly expensive validity checks or heuristics. As Togelius, Justinussen and Hartzen (2012) show however, search-based approaches can still be suitable for selecting appropriate parameters for an ASP-powered generator; alternatively parameters could be provided by designers based on expert knowledge about the particular domain of interest.

### 3.3.1 ASP for games and puzzles

Smith and Bryson (2014) describe a system using ASP to assemble pre-generated room modules and their variants into a consistent dungeon layout according to connectivity, and suggest methods for hierarchical refinement of key locations.

However, sufficiently complete descriptions of complex domains lead to large and slow problem representations, so it may be appropriate to split the encoding into multiple problems (Smith et al., 2012) or use an automated approach for ASP code authoring.

A similar approach is taken by Neufeld, Mostaghim and Perez-Liebana (2015), who use an underlying base problem definition combined with some subset of possible extension fragments to generate game levels based on a combination of the formal specification in the VGDL input and parameter evolution. Some extension fragments are selected automatically based on the presence of certain features in the specification, whilst others are set based on sensible limits, which may be mutated. As in Togelius, Justinussen and Hartzen (2012), a search-based approach is used to evaluate various parameter assignments.

Togelius, Justinussen and Hartzen (2012) make use of a comparatively simple dungeon/-cavern generator implemented in ASP as part of a larger multilevel generation system. The ASP portion of the system generates organic dungeons populated with enemies and special tiles that must be reached in order to unlock an exit, using constraints that ensure no simple

route to completion exists. A number of important parameters for generation are exposed to a population-based genetic algorithm, which searches the space of possible parameter assignments using a paired-agent fitness approach as described in Sec. 2.5.1. For parameter assignments that lead to multiple answer sets an average of the agent assessments is taken, although the authors note that variance is often high in these cases. The use of a genetic algorithm to tune parameters allows rapid search of a wider space of possible values than would typically be feasible under the combinatorial approach used by ASP alone, however as there is no guarantee that the entire space is covered it is entirely possible that desirable but narrow local or even global maxima could be missed. This drawback is potentially less significant, given that the goal of the approach is to produce a diverse variety of interesting and challenging output, rather than maximise for any particular optimum result.

#### Anza Island

In *Anza Island*, Compton, Smith and Mateas (2012) present a system which allows the player to indirectly author ASP as part of gameplay, producing a partially externally-specified ASP program for content generation. As play progresses, the player gains access to additional constraints that may be added to the base problem encoding in order to influence the emitted answer sets — the actual program used to re-generate the game play environment is therefore a composition between the underlying description and the user-provided fragments. The gerator's output is a graph of navigable connections, using player-specified constraints such as "Monumental Stone Head can't be connected to Hidden Grotto". The graph generation problem — including player constraints — is formulated in ASP and solved again each turn at runtime to update the game map with valid connections between landmarks. In this context, the additional elements are in the form of the player-selected constraints, that are used and varied turn to turn to restrict the selection of possible answer sets to only those that might be helpful to the player. The ASP program for *Anza Island* is therefore composed of three main pieces: the basic facts about the island and rules about how pathways connect; the additional player-selected constraints; and additional constraints that represent the 'mood' of the system, i.e. by making the longest path possible, or maximising for the number of connected monsters. After each turn, it is possible that the mood or the player selections may have changed, leading to production of entirely different answer sets and thereby altering content in-game.

#### Warzone Map Tools

The first significant use of ASP for game content production was the *DIORAMA* Real-Time Strategy (RTS) map generation system produced as part of the *Warzone Map Tools* project[3] by M. Brain and F. Schanda, and described more fully in Smith and Mateas (2011). As with the Sentient Sketchpad system (Liapis, Yannakakis and Togelius (2013a), Sec. 2.3), the generation approach initially considers a low resolution 'sketch' of the final map, where ASP is used to place player bases, unaligned resource nodes and strategic height variations according to a number of

[3] `http://warzone2100.org.uk/` – accessed 25 November 2020

customisable constraints. A range of non-ASP domain-specific decoration techniques are then used to convert the low resolution sketch into a playable map, where an optional second ASP process may be used to define appropriate configurations for complex player base layouts based upon surrounding terrain affordances. Overall this system is highly tailored to the requirements of the *Warzone 2100 game*, though a number of tweakable parameters are exposed to the user via GUI. The initial sketch generation approach may be generalisable to other RTS games via customisation of the domain-specific conversion methods, without significant change to the underlying ASP program.

#### Refraction

This early academic investigation of ASP for game content production was performed by Smith and Mateas (2011), who suggest the use of ASP as part of a design space approach to content generation, given that the declarative nature of the approach allows developers to focus more on 'what' should be generated rather than 'how'. They provide a general approach for mapping PCG problems to answer set programming, and note that the process of developing an ASP-based generator is often informed by the properties of artefacts output at intermediate stages, which leads to a responsive 'sculpting' approach to successively constraining the viable output space throughout development (Smith, 2012).

These approaches are put into practice for the development of 'Refraction', a fraction-based educational puzzle game. Smith et al. (2012) describe an original implementation of the system, which uses a comparatively simple generate-and-test approach to produce abstract specifications for individual puzzles, followed by an application of depth-first search (DFS) to attempt to find a valid solution for a given specification, and another iteration of DFS to produce a physically feasible 2D layout for identified solutions. Initial attempts to improve the system performance involved applying an ASP-inspired geometric restart strategy to the DFS stages, before replacing each DFS wave and even the overall generate-and-test with ASP re-implementations. They observe that the implementation lines-of-code metrics are an order of magnitude smaller for the ASP implementation than the original Java approach, and that ASP is able to provide solutions in less than a second for certain complex problems that the DFS was unable to find in over an hour. In this case, the domain space is necessarily defined by the puzzle solution rules, as encoded via ASP, however the overall approach (generate a variety of abstract puzzle specifications, identify solutions for those puzzles, embed those solutions in a viable representation) could easily be converted to any puzzle type for which an ASP solution approach is available (e.g. any so detailed by Caylı et al. (2007)).

#### Infinite Refraction

An extension of the work by Smith, Butler and Popović (2013) is based upon the observation that interesting puzzles necessarily have no shortcuts that admit trivial solutions. A modification to the original Refraction system is presented which is able to quickly identify solvable puzzles without shortcuts in a combinatorial space too large to feasibly check by hand. Addi-

tionally, the system is able to require the presence of certain concepts in the produced puzzles, which aids the selection of appropriate puzzles for educational purposes. The modified system uses *metasp* — an optimisation approach implemented in ASP — to enforce reference solution minimality (that is, ensure that for any given solution to a particular puzzle, no solution to the same puzzle exists that relies only on simpler concepts). In some senses, this new system is more domain-specific than the original implementation, as it relies on specific properties of the type of puzzle being generated. However, the overall technique (metasp optimisation over answer set properties) is conceivably generalisable as an approach to providing a range of preference options over answer sets.

#### Compositional

Togelius, Justinussen and Hartzen (2012) make use of a comparatively simple dungeon/cavern generator implemented in ASP as part of a larger multilevel generation system. The ASP program itself generates organic dungeons populated with enemies and special tiles that must be reached in order to unlock an exit, using constraints that ensure no simple route to completion exists. A number of important parameters for generation are exposed to a population-based genetic algorithm, which searches the space of possible parameter assignments using a paired-agent fitness approach as described in Sec. 2.5.1. For parameter assignments that lead to multiple answer sets an average of the agent assessments is taken, although the authors note that variance is often high in these cases. The use of a genetic algorithm to tune parameters allows rapid search of a wider space of possible values than would typically be feasible under the combinatorial approach used by ASP alone, however as there is no guarantee that the entire space is covered it is entirely possible that desirable but narrow local or even global maxima could be missed. This drawback is potentially less significant, given that the goal of the approach is to produce a diverse variety of interesting and challenging output, rather than maximise for any particular optimum result.

Another approach to using ASP for PCG is as a compilation mechanic for a library of pre-produced content elements, in order to ensure that certain properties (such as acyclicity or zero empty dead ends) are present in the generated output. Smith and Bryson (2014) present a system for generating rogue-like dungeons that uses an annotated library of hand-authored room and corridor chunks, combined with a hierarchical approach to generation in order to mitigate the potential combinatorial explosion. The 2D area to be generated is divided into a grid where cells match the chunk size of the library content. Each cell is allocated a chunk, where manually duplicated and rotated chunks represent possible variations for chunks without rotational symmetry. Potential solutions where corridors do not line up are discarded by appropriate constraints, and a few additional properties such as presence of a suitable number of (tagged) special rooms are checked. This phase is then potentially followed by a separate step performing procedural decoration and population of the dungeon. Elements of the system's domain are specified by a combination of hand-authored chunk content and appropriate properties encoded directly in ASP. This is simple enough for constrained problems, but would

not scale well and could become unwieldy if the library of available content grew significantly. The system would also require significant re-engineering in order to repurpose it for any other type of domain.

### 3.3.2 Non-PCG applications in games

ASP has also been applied to a small number of games-related problems that do not explicitly relate to the generation of content in the usual sense, but nevertheless demonstrate applicability of the technique to domains with game-like properties (continuous time/angles/inputs).

Calimeri et al. (2013) consider the use of ASP as part of a planning algorithm for an artificially intelligent game playing agent for the game 'Angry Birds'[4].They use the `dlvhex` system (see Sec. 4.5.1), which allows for the contents of atoms to be populated via calls to external computation: in this instance, physics calculations that estimate outcomes and query for the success of iterative attempts at solving a given puzzle with varying parameters. Although this approach does not generate content for the given game, it demonstrates the potential for ASP-related approaches to be used for quantitative assessment of generated content as in Togelius, Justinussen and Hartzen (2012) (Sec. 2.5.1); alternatively, it indicates the possibility for ASP driven opponents to be presented as an interactive feature in suitable games.

Even for games where the overall domain is unsuitable for constraint-based approaches, due to e.g. the size of the domain or representation difficulties, it may still be the case that game-playing agents are able to make use of ASP to select appropriate solutions for certain sub-problems. Čertický (2013) details an approach to modelling vulnerable areas around a player's base in a popular real-time strategy game (Starcraft)[5] in ASP. When combined with suitable facts about the structures available and the relative sizes of each players' units, this allows for the selection of appropriate construction orders to ensure that the base is well-defended. This is analogous to the Warzone 2100 base construction step (above, 3.3.1), although performed by an agent during the runtime of the game, and with the explicit intention of blocking access to the base rather than ensuring easy access throughout.

### 3.3.3 ASP for mixed-initiative content production

Given a sufficiently complete set of restrictions on the output space, ASP can also assess manually edited or externally provided content in order to highlight constraint violations and suggest corrections, leading to a mixed-initiative content production approach (Boenn et al., 2011; Karavolos, Bouwer and Bidarra, 2015; Butler et al., 2013).

*Anza Island* by Compton, Smith and Mateas (2012) presents a card-based interface for mixed-initiative ASP co-authoring as a game mechanic, where players must compose additional constraints from cards with ASP fragments found during gameplay. These custom constraints are used to augment the (re-)generation of the play space.

[4] `http://aibirds.org/` – accessed 16 November 2020
[5] `https://starcraft.com/en-gb/` – accessed 19 November 2020

One advantage of ASP is that it is able to treat externally-specified facts and internally-inferred facts in an identical manner. This means that for a sufficiently well-specified domain, a partial or incomplete solution may easily be extended to a complete one using the same mechanism as solving from scratch (under the assumption that the partial specification is internally valid).

One generative ASP system which makes use of this property is *Anton* (Boenn et al., 2011), a computer-aided composition tool for working with musical pieces in the Renaissance Counterpoint style. This particular musical style is sufficiently well formalised to be amenable to reasoning over via ASP. Given basic information about the desired length and the musical key, *Anton* is able to use an ASP encoding of the Renaissance Counterpoint composition rules to simultaneously generate both a novel melody and a harmonisation for a short piece. Alternatively, given partial information (just a melody, or just the first few bars) the system can equally provide an accurate completion; automating part of the process and allowing composer and system to share the burden of content production. Finally, with a few extra rules, *Anton* is also able to diagnose and highlight rule violations in an externally specified piece, making it useful for automated assessment of independent work. However, the authors note that a number of additional challenges relate to the composition of larger pieces, most notably the problem of supplying both variety over the length of the piece, and an overarching global structure.

As described in Sec. 2.3.1, *Anza Island* (Compton, Smith and Mateas, 2012) also makes use of a partially externally-specified ASP program for content generation. In this context, the additional elements are in the form of player-selected constraints, that are used and varied during runtime to restrict the selection of possible answer sets to only those that might be helpful to the player. The ASP program for *Anza Island* is therefore composed of three main pieces: the basic facts about the island and rules about how pathways connect; the additional player-selected constraints; and additional constraints that represent the 'mood' of the system, i.e. by making the longest path possible, or maximising for the number of connected monsters. After each turn, it is possible that the mood or the player selections may have changed, leading to production of entirely different answer sets and thereby altering content in-game.

### 3.3.4 Authoring ASP

A potential issue with ASP is the close integration between the rules that specify the 'shape' of a particular problem, and the specification of facts relating to a particular instance of that problem. Some solvers support constant declarations that allows passing the value of certain constants to the program during invocation, but this can become unwieldy for large numbers of variables. For small problems, it is convenient to specify facts in the same file and format as the rules, and modern solvers also support loading multiple input files, allowing facts to be specified independently of the program. This can still be labour-intensive for large problems though, and so a number of alternative approaches have also been used, as detailed below.

In the automatic progression system developed for *Infinite Refraction*, Butler et al. (2015) present a multi-stage ASP system. First, a comparatively simple ASP program generates an

output file specifying all possible desired configurations for levels, where a configuration is a high-level description of a level, detailing only the number of desired elements of each type or concept. These configurations are later used as input to a more complex program which uses the methods as described in Smith, Butler and Popović (2013) to produce complete puzzle descriptions. This approach is comparable to the one used by Togelius, Justinussen and Hartzen (2012), where instead of a preliminary ASP program a genetic algorithm is used to provide a range of parameter configurations to a more fully-featured generator, guided by domain-specific heuristics on applicable metrics.

To support the use of ASP for reasoning about institutions, Cliffe (2007) provide a translation process from a DSL, Inst$AL$, to an ASP model. Inst$AL$ is an 'action language' designed to model the progression of events in a domain over time, and as such is semantically closer to the methods used to model institutions. It is designed to be human-readable and succinct, and can be automatically translated into a formally-equivalent ASP model, in order to reduce the burden of authorship and eliminate the potential for programming errors. The Inst$AL$ syntax and file format are both highly constrained, and tailored to nature of the institutional modelling problems being considered. This simplifies the process of verifying and translating the specifications, at the cost of restricting the capabilities of the language.

In contrast, Gaggl, Schweizer and Rudolph (2015) take an alternative approach, of providing a formal translation to ASP of a domain-general knowledge representation language: OWL (Sec. 7.2.3). By considering only finite knowledge bases with a domain defined by known individuals ('bounded models'), it becomes possible to show that the answer sets produced by a model translated from a given knowledge base correspond exactly to the valid interpretations of that knowledge base. This means both that it is possible to use efficient answer set solvers for OWL reasoning tasks such as satisfaction, but also that entity information and relations stored in an OWL knowledge base may be imported into an ASP format. As OWL is intended as a domain-agnostic knowledge representation format, the translation process specified should be applicable to a more general class of problems than the institution-specific Inst$AL$ language.

For small problems, in-program specification of facts and constants is manageable and simple, but for large or complex domains it appears advantageous to provide a mechanism for automatic provision of these data, whether generated algorithmically (Butler et al., 2015; Dahlskog, Togelius and Nelson, 2014), or translated from some external source (Cliffe, 2007; Gaggl, Schweizer and Rudolph, 2015).

Following the discussion of data-driven ASP authoring above, we recognise that complex generation domains are likely to require augmented ASP authoring approaches. One of the key benefits of ASP for PCG is the ability to consider a problem in terms of its possible design space; however languages like Inst$AL$ (Cliffe, 2007) and VGDL (as in Neufeld, Mostaghim and Perez-Liebana (2015)) indicate that ASP may not be the only possible design space description language choice. Formal description of a design space may potentially be undertaken in a description logic such as the OWL, and then translated into ASP in order to perform generation within that space (Gaggl, Schweizer and Rudolph, 2015).

One possible advantage of working within an existing description language is that tools and

libraries are already available, another is that there may be applicable techniques described in existing literature. As examples, OWL has several existing editors and reasoners, and provides support for inheriting concepts and individuals from shared parent ontologies, of which many already exist. Also, representation of the design space of desired content can allow for useful preprocessing and/or partitioning before ASP generation. A large, complex problem may be split into several separate stages, in order to initially produce an abstract model of the desired content, and then iteratively refine it towards a finished, usable output. A number of existing generation approaches make use of a similar technique - most commonly by generating a mission representation as a skeleton for a level, and then producing a playable space within which to embed it (Lavender and Thompson, 2016).

A few domain specific languages (DSLs) for game production already exist, most notably Puzzlescript[6] and VGDL (as in Perez-Liebana et al., 2019; Neufeld, Mostaghim and Perez-Liebana, 2015). 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 (Antoniou and Van Harmelen, 2004) 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.

## 3.4 Discussion

Games are generally known environments with explicit, finite domains of relevant content that may be reasoned over, though these ontologies are ultimately often only defined informally and implicitly through the presence of assets, imperative code and designer mental models. ASP is particularly well-suited for structured generation tasks within these finite domains, but requires explicit formulation of the problem spaces in a language and style that may be unfamiliar to many designers and developers. Even in the case of continuous content such as terrains or procedural variance, ASP may be useful for generating high-level or 'abstract' specifications or sketchs that can then be used to guide or constrain downstream processes.

In addition to constraints concerning the geometry and visual appearance of generated content, it may be appropriate to model and reason over the play experience and skills development of the hypothetical player (Deterding, 2013). One possible approach is to integrate a semantic model of the 'narrative' to guide successive, iterative generation steps; using a range of smaller ASP programs that gradually refine an abstract model of the generated content into the final output. Dormans (2011) describes a similar, graph-grammar-based approach using rewrite systems to iteratively refine a model of the desired content, prior to translation to a usable play space.

Modern ASP solvers such as clingo are capable of producing answer sets to a well-defined problem very quickly, making ASP-driven generators potentially appropriate for domains in-

[6] `https://www.puzzlescript.net/` – Puzzlescript, accessed 19 September 2020

cluding realtime generation and structured graph-generation. In the following chapters we present two systems developed under this paradigm — one realtime, one structured; and we highlight the application of Smith and Mateas's (2011) design-space assessment concept to inform development appropriate for our industry contexts.

# Chapter 4

# Wave-based Combat

In this chapter we present an initial investigation into the use of Answer Set Programming (ASP) to generate structured content specifications: specifically, wave-based combat (WBC) progressions. This is an application of the constraint-based approach outlined in the previous chapter and introduced by Smith and Mateas (2011) to an initial, constrained problem in a new domain. The approach is a practical application of the technique to generate structured content in a space with sufficient variance that we can 'sculpt' it according to design intentions, and provides a tangible, applicable benefit with relevance to commercial game development.

The problem of generating appropriate wave-based combat specifications is an instance of iteratively selecting from a loosely combinatorial space of options according to a range of both hard gameplay constraints and softer design-driven desiderata, and is therefore well-suited for constraint-driven generation approaches. We illustrate a series of encodings of the generation problem in ASP, whose outputs are sets of facts that each represent a self-consistent valid answer to the problem as specified, and investigate the salient features of specimen outputs from those encodings. We observe that during development of the system, we may observe that the problem is incorrectly specified in some particular aspect — however the flexibility of the declarative approach can greatly reduce the (re-)engineering burden associated with pivoting to a new formulation (as covered in Smith's thesis), and we demonstrate a number of these repairs.

We begin by discussing the concept of combat waves, progression and their broader applicability in games, and considering some of the relevant assumptions we may make based on common vocabulary and expectations in the design space (Cheng (2017), Sec. 4.1, Sec. 4.1.1). Then we delve into the implementation details of our approach, considering the information we have available to us and present an initial solution to producing progressions (Sec. 4.2, Sec. 4.2.1, Sec. 4.2.2). This initial approach is refined to provide a more sophisticated model of the problem; we then discuss some of the sample outputs from this system, and discuss some of the new features it can support (Sec. 4.2.3, Sec. 4.3). There are a range of design implications and new possibilities introduced by this approach, including the different points at which the system may be used: design time, static generation for variance, or dynamically,

which we address with reference to both the subfield of Procedural Content Generation (PCG) design patterns (Smith, 2014; Cook et al., 2016) and the field of player modelling (Smith et al., 2011; Boulton et al., 2017)(Sec. 4.4, Sec. 4.4.1). We introduce the skills-acquisition model, relate it to the current context, and explain how it could lead to a more complex modelling problem (Sec. 4.4.2). We discuss practical considerations including choice of ASP solver and a mechanism to translate output specifications into content as part of integration with an existing game and engine (Sec. 4.5, Sec. 4.5.2). In conclusion, the use of ASP presents a viable, flexible approach to generating combat wave progressions that allows for both easy declarative sculpting and some exciting new generativity-based opportunities (Sec. 4.6).

## 4.1 Wave-based combat in games

In this section we motivate the problem by discussing the definitions of and issues relating to combat progression design, and detailing the context and assumptions relevant to our approach. We provide definitions of several of the common terms and concepts relating to wave-based combat, introduce the broader issue of progression design for action-adventure levels and more-linear portions of open-world games such as dungeons, and explain the Arena setting for wave-based combat progressions. We follow this with consideration of some issues and opportunities with the current, generally hand-crafted and eyeballed for correctness, approach to producing progressions, which motivate our intention to develop a system for generating wave progressions automatically. In Sec. 4.1.1, we justify restricting our scope to specifically arena-based / wave-based combat, and discuss two comparable commercial datapoints. We finish with an overview of how this informs our specific assumptions, and some of the interesting implied constraints that the context presents.

Combat in some form is a common element of many games. A typical experience of many 2- and 3-D action-adventure games is that a player, and perhaps allies, participate in combat encounters in a virtual space, interspersed with non-combat elements such as puzzles, narrative scenes and other challenges. Encounters start when a player engages with enemies — either through choice or because they are attacked — and typically end when the player has defeated all enemies. The failure state where the player instead is defeated is mostly considered an undesirable outcome by both players and designers alike. A play session is likely to include multiple such instances of combat, which depending on the game, may be in an explicitly designed ordered progression, or whose order may arise organically as a result of player freedom to explore a more 'open' game world. Though the specifics of combat encounters and the ways in which they are resolved vary between games and genres, some common design elements, constraints and requirements are shared, for example:

– subsequent encounters are frequently within a similar band of desired difficulty to avoid suddenly becoming too challenging or too easy, whilst also tending to increase in difficulty or complexity overall over time in order to give players the opportunity to develop skills;

– designers typically have access to a finite array of variable elements to adjust to make each

subsequent encounter unique, and aim to hit a 'sweet spot' between variance and coherence to preserve player interest without breaking suspension of disbelief; [1]

– in normal gameplay there are often narrative constraints on which kinds of enemies and combat locations are likely to be present or available at different points in the game.

In general, the production of a reasonable and enjoyable succession of combat encounters is an important design task commonly necessary for game development, with elements of authorial intent in terms of intended difficulty progression, provision of suitable opportunities for the player to learn and develop gameplay skills, and constraints relating to narrative elements and a necessarily limited palette of enemy archetypes and combat locations.

To establish a set of common terms of reference, through the rest of the thesis we will use the following definitions — which are commonly applicable across many games and genres.

**Enemy:** in many games: a recognisably distinct generally-mobile hostile entity with a finite health pool and one or more ways to damage or impede the player, and is counted as defeated when that health pool is reduced to zero. For pragmatic reasons relating to both legibility of gameplay and to the development costs of art, animation assets, and coding for behaviours, generally 'enemy' entities in a single game belong to one of a small number of distinct archetypes. These consist of a distinct combination of appearance, general power (health/damage levels), special abilities, and other properties that affect how the entities interact with the game world and the player. These archetypes serve specific design purposes and roles in combat, and commonalities in appearance communicate learnable information for the player(s). Though there may be slight variations in power or abilities or cosmetic appearance within a single archetype, enemies of that type are generally united by a broadly-consistent visual style and behaviour.

**Group:** for our purposes we restrict the term group to refer to only one or more enemies *of a single type*. Enemies of different types may require different skills or approaches to defeat, but it is generally safe to assume that if the player possess the skill or ability to defeat a single enemy of a particular type, a similar approach will also serve for additional enemies of the same type. Depending on the game it is also more likely that enemies within a group of the same type will share or compete for relevant resources, including abstract resources such as proximity to the player character, than between enemies of differing types.

**Wave:** a wave consists of one or more groups of enemies that are encountered simultaneously. During a single wave, the player(s) may generally dispatch enemies in any order — prioritisation is a gameplay skill and an open choice presented to the player — and depending on the game a single 'wave' of enemies may last until all are defeated, or subsequent waves may be introduced based on a specific measure of time elapsed or other triggers such as an un-interrupted call for reinforcement.

[1] for more on 'knob-based design', see `https://magic.wizards.com/en/articles/archive/making-magic/more-stories-city-2018-10-01` — accessed 25 November 2020

**Encounter:** a single encounter lasts from the beginning to the end of a period of combat, which may consist of only one wave of enemies or multiple. The concept of an 'encounter' frequently also describes several contextual elements that can inform development and help to differentiate successive encounters that may otherwise contain very similar enemy selections. The location where the combat occurs may have significant combat implications, such as reduced visibility due to e.g. smoke; terrain hazards; destructible cover; or obstacles such as crates, barrels or pallets. An encounter has both a narrative and a gameplay purpose for the combat, e.g. 'fight off the pirate ambush' and 'introduce the player to the abilities of the basic Pirate type enemy', and the intended reward the player will gain from successfully completing the encounter e.g. 'the opportunity to continue journeying to the Docks'.

**Progression:** within a game or single gameplay session, we use the term 'progression' to denote an ordered series of combat encounters. In linear games the specific order in which players will progress through available content can be assured in advance, while even in non-linear or 'open world' games there is frequently an expected partial ordering based on intentionally gated chokepoints in the player experience, or even specific linear subportions of those worlds, such as 'dungeons' (see Chapter 5). A progression is generally deliberately designed — to communicate a particular narrative, stretch the player to develop their skills, provide a varied and challenging combat experience, or frequently all of the above.

One of the key design considerations for a satisfying progression is the development and variance of overall difficulty of the encounters throughout, in a manner appropriate for the current player's skill. This is challenging in practice as abstract 'difficulty' is notoriously tough to measure objectively, and yet ideally the the challenges 'are always at the margin of our [the players'] ability' (Koster, 2005). The subjective difficulty of a given wave is dependent on a range of factors, including the character abilities or weapons that the player has unlocked, the degree to which the player has mastered those abilities, the composition of the wave in terms of the number and varieties of enemies present, and even potentially the precise relative locations of those enemies. Different kinds of enemy will move at different speeds and in differing ways, attempt to injure or affect the player character with varying attacks, and have a range of health levels — meaning that different tactics are generally suited to defeat each differing type. The presence of certain combinations of enemies may complicate the selection of appropriate tactics: it may be risky to stand still and use ranged attacks against a distant enemy, while being harassed by otherwise weak ground troops, or it could be difficult to safely evade a powerful but slow enemy in the presence of aerial assault. Fortunately, a precise measure of difficulty is rarely necessary and for many game contexts a reasonable heuristic can be found to guide development of a satisfying progression — though there are frequently additional narrative or gameplay constraints that must be also be considered.

**Arena:** some games include (either as additional post-game content or embedded within the narrative) an arena-style challenge consisting purely of many successive waves of enemies

with no internal narrative component, and all weapons and abilities unlocked from the start. One advantage of this kind of pure-combat content is that it provides a unique opportunity to explicitly challenge the player with a consistent, known set of abilities and locations without requiring many additional narrative or environmental assets. These are often produced manually by designers and tuned based on user testing until they 'feel right' — provide a satisfying sense of progression between waves and sufficient degree of challenge, with variance in wave compositions to sustain player interest, and an overall culmination towards a finale that provides a sense of achievement.

**Landmark wave:** a wave that is deliberately and notably different from the ones preceding and following it. In an arena context without changes to location or narrative, successive waves can become monotonous. A common practice is to introduce occasional milestones in the form of waves that break expectations in some ways. Where they are easier than the general trend these are sometimes known as 'palate cleansers', and occasionally include other temporary changes such as novel environmental hazards or an opportunity to break pace. Alternatively instead of the expected group(s) of enemies there may be a single custom enemy with increased health, specific challenging attack patterns or obscure weaknesses; a 'Boss' enemy/wave. These are typically significantly more challenging.

> "The point of game AI is not to win against the player, but to lose in style."
> Brian Schwab, Turing Tantrums GDC'11 (Schwab et al., 2011)

Though narrative and gameplay constraints are reduced or eliminated by the framing, the development of arena style wave-based challenges can be a slow and laborious process requiring many iterations of testing and tuning. An automated system capable of generating one or more suitable progression(s) of waves could alter the nature of the labour required for these development tasks, promoting exploration of possible alternate configurations over rote adjustments to spreadsheet datafiles. The benefits would be particularly felt in scenarios where the challenge experience is being developed prior to or in parallel with the process of rebalancing or tuning player and enemy abilities that frequently occurs towards the end of game development, which can invalidate and necessitate reconstruction of previously acceptable hand-designed progressions. An automated system would also open new avenues for design exploration as in Smith's (2014) paper on unpacking the design impacts of PCG in games, specifically the qualities of *reacting in a surprising environment* and *practising in different environments*. Varying the order and makeup of the combat waves reduces the benefit that can be gained from rote memorisation and preparation, forcing players to be more adaptive and responsive; and providing a system that can change up the wave compositions means that players have access to a more varied pool of possible combinations than the fixed number of hand-authored waves of present systems. However, the most novel outcomes could be achieved by focusing on another property definition within that paper: *building generator strategies*. For example: instead of pre-defining combat progressions at design-time, an automated generation system could potentially produce successive waves during runtime based on player performance, tailoring the difficulty experience

according to appropriate designer-guided responses. This is an outcome that could not be replicated simply by providing larger volumes of hand-authored content, and has the potential to lead to new forms of play where the player is aware of the generator and its intended responses, and makes deliberately triggering and guiding those responses a core intention of their actions. In the remainder of this chapter we discuss our implementation of an automated system for generation of wave progressions using ASP, and the implications and considerations that arise from our work.

### 4.1.1 Problem context

Arena style wave-based combat progressions are a prime candidate for procedural generation approaches. The diegetic reduction of the moment-to-moment gameplay experience to purely mechanical fighting eliminates the need for many of the difficult-to-generate narrative considerations and environmental aspects that are common in the main game flow of many games. In addition, the problem space is frequently constrained by the quantity of enemy and environment types produced for the main game. A baseline manual approach could consist of combinatorial combinations of enemy groups and simple difficulty ramping, given an appropriate heuristic for estimating individual difficulties and successive wave triggers (Cheng, 2017). We can make use of several of the implicitly-understood limitations that such an approach encodes to inform a constraint-based generation approach, such as on maximum number of enemies in a wave, or maximum number of enemy varieties in a single wave. As discussed in Sec. 4.3.2 there is also potential for broader and more powerful constraints relating to both designer intent and player modelling. In the remainder of this section we discuss how approaches developed for arena-based WBC might be more broadly applicable to general game combat progressions, illustrate some specific assumptions we make in our own implementation and their roots in existing domain artefacts, and provide an initial suggestion for the benefits that this approach holds over manual authoring.

An automated system for the controllable generation of wave-based non-narrative combat progressions is a step towards generation of combat progressions more generally. The combat progressions in levels, or whole games, have several similarities to those required for arena-style combat — they both typically require variety, broadly increasing difficulty, and work within a limited palette of enemy types and locations. The two primary differences are narrative, and mechanical assumptions. Arena progressions tend to have minimal or no internal narrative; each 'encounter' takes place in the same location, and differs only on the number and composition of enemy groups present. Mechanically, arena progressions typically assume that the player already has access to and is familiar with how to use all available player character abilities, while level- and game-progressions must support the introduction of and training for each of those abilities (Koster, 2005), and similarly while arena progressions may make use of any or all enemy types, the broader progressions are constrained by both narrative context and the necessity to gradually introduce new enemy types. Table 4.1 provides an overview of some of the commonly-understood different and similar elements between different scopes of combat

progression present in many action-adventure games.

| Differences: | **Arena** | **Level/Dungeon** | **Whole Game** |
|---|---|---|---|
| Enemy types | Fixed (all types) | Local palette | Changing throughout |
| Player abilities | Fixed (all abilities) | Limited by progress | Accumulating throughout |
| Environment type | Fixed | Local palette | Changing throughout |
| Narrative elements | None | Scene / plot point | Complete narrative |
| Skill introductions | None | Local concept | Ongoing throughout |
| Exploration | None | Mostly linear | Design dependent |
| Interstitials | 'Palate cleansers' | Other challenge kinds | Design dependent |
| Rewards: | | | |
| Intermediate | Next wave | Power increase | Power, narrative payoff |
| Overall | Completion | Narrative, next level | Completion |
| Similarities: | | | |
| Variety | Broad | Yes | Broad |
| Difficulty | Increasing | Increasing | Increasing |
| 'Boss' fights | Mid & Finale | Mid & Finale | Per-level & Finale |

Table 4.1: Broad typical similarities and differences between different progression scopes.

In an arena context, the difficulty of a given wave serves primarily as a challenge to overcome in order to reach the next wave, and so on until completion of the challenge. In contrast, the difficulty of wave progressions within game or level are often tuned to serve several additional purposes. For example, new and challenging enemies are often introduced for the first time immediately before the player will gain access to a new weapon or ability, and then re-introduced once the player has had a chance to practice the new skill — the initial fight is designed to be challenging for the player in order to highlight the comparative ease with which the same enemy may later be dispatched once the skill is mastered. In both kinds of progressions, wave compositions and difficulties are deliberately varied in order to provide satisfying pacing to the combat, often following a rough arc that culminates in a challenging 'boss battle' at the end of individual segments. An automated system that could generate these varied compositions would be suitable for arena-style progressions, and with further development of systems for skill acquisition and narrative/environmental context could be part of a system for level or whole game combat progression generation.

The generation of an appropriate wave-based combat progression is an interesting structured generation problem due to the presence of arbitrary gameplay and design constraints, and has been comparatively under-explored academically. The genres of games for which the problem is relevant are ones that are not as well-represented in existing literature, despite the domain seeming well-suited to generative approaches according to the criteria proposed by Grey (2017). Horswill and Foged (2012) present a numerical-constraint-solver driven system for populating roguelike dungeons with enemies that is similar given the parallels between the general level population problem and the related non-narrative progressions for arenas, however their numeric approach considers only the presence or absence of monsters in a given location, and

the player's expected health delta as a result of those interactions. Cheng (2017) describes a number of approaches for determining when to present the player with the next wave, which are complementary to the approach we present here.

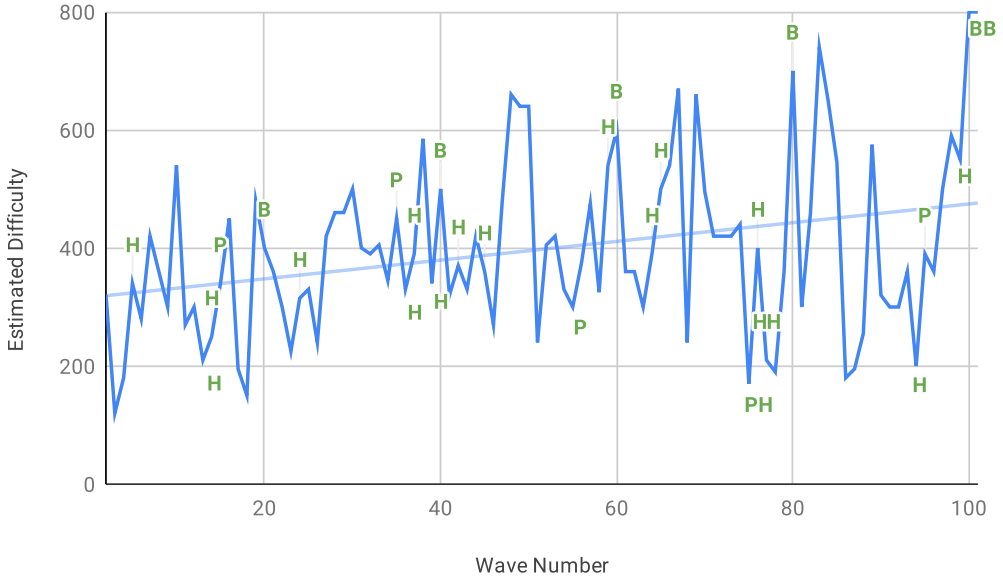



Figure 4-1: Progression of wave difficulties in an existing non-narrative combat challenge.

To facilitate production of a generative system we take our cues from an existing context and constrain our scope accordingly. One of the advantages of selecting an existing commercial game for a testbed is the fact that all game systems and content aside from the generation component are already present, reducing the amount of additional work needed to test usable output. Other benefits include working within genuine industry constraints, known to be practical in a shipped game; access to designer insight relating to this context, and the presence of two extant designer-produced combat progressions for comparative evaluation work. Finally, the most valuable aspect is the assistance of the original designers who produced the content and systems, who are able to provide guidance and domain insights towards production of an automated system.

This context allows us to proceed under the following assumptions: for the purposes of the progression generation, we are able to ignore all narrative considerations and potential environmental effects. To facilitate comparison, and in accordance with the reference game's design, we make use of thirteen distinct enemy types (see Listing 4.2), with associated estimated difficulty ratings and expected group sizes. In our reference progressions, there are either sixty or one hundred and one waves, and due to engine and design limitations there are at most ten enemies at once of up to three different kinds per wave. As in Table 4.1, progressions of wave difficulties in these experiences follow a different set of requirements to those in the main game. As these arena progressions are only unlockable once the main game has been completed, it is

expected that players will already be familiar with the full range of weapons, abilities, enemies and appropriate tactics, and therefore the focus is more on providing a challenging and varied experience than on introducing new concepts to the player. Fig. 4-2 shows a designer-estimated difficulty for each of the sixty waves in one such progression, and Fig. 4-1 provides another, longer example of a designer-authored progression.

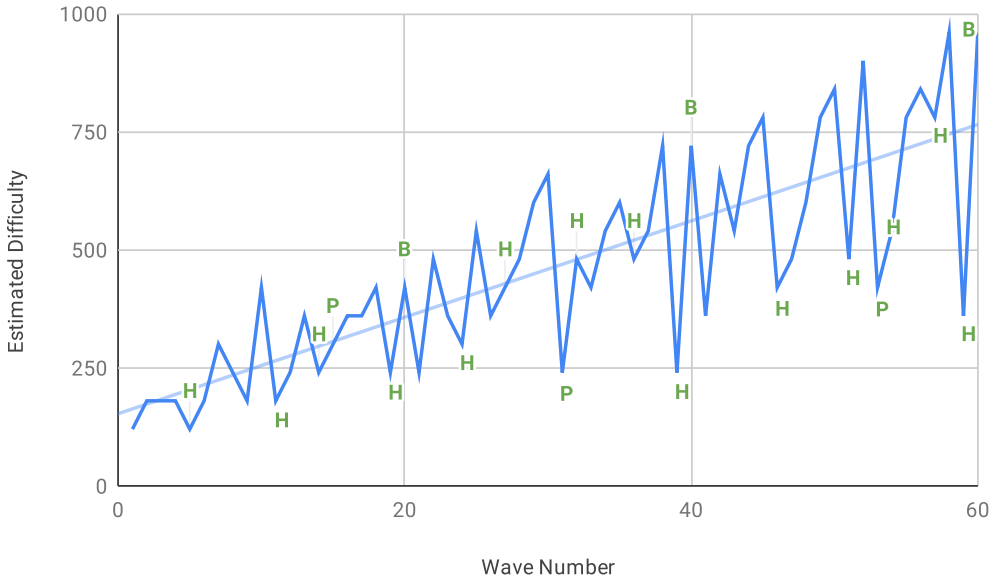



Figure 4-2: Progression of wave difficulties in a shorter combat challenge in the same game.

While there is an observable trend towards increased difficulty in later waves, the delta for any given next wave is highly variable. In addition to the normal waves, composed of enemy groups as described above, there are also 'landmark' waves that help to vary the experience and signal intermediate milestone achievements. In this progression, Boss battles (B) are presented after every 20 waves — these are unique experiences that require different tactics to typical waves, and in this example are drawn from boss enemies developed for the games' narrative mode. Roughly half-way between each boss are 'palate-cleanser' (P) waves: unusually easy waves that serve to temporarily break the pace while also introducing unique environmental effects, often as preparation for the following boss battle. Finally, waves marked H include enemies that restore health to the player when they are defeated — these are generally found immediately prior to boss battles or other difficult waves, to signal that a challenge approaches and provide the player with a buffer of bonus health. These alternate wave types provide an element of higher-level structure to the player's experience of these specific progressions, but are not present in all wave progressions across genres. Regardless of whether landmark waves are present, the groups of enemies for all other waves must still be suitably generated.

Game difficulty is hard to judge objectively, however in order to provide a satisfyingly varied difficulty curve it is still necessary to reason at design-time over some heuristic stand-in for the

expected difficulty of a given wave composition. The simplest approach is to assign each enemy archetype a single relative difficulty score. This can be done either algorithmically according to properties like health or damage or, where applicable, experience or other rewards granted, or manually according to designer knowledge. Any changes to the archetypes' properties or abilities will require updating this estimated value, and without some automated system for generating wave progressions would generally also require manually rebalancing or rebuilding any previously authored progressions that relied on the prior values. A naive approach to calculating wave difficulties would then be a simple sum over all the difficulty scores of all the entities in the wave. More sophisticated representations are possible (see Sec. 4.4.2), but as an initial approach an unaugmented sum of estimated difficulties already suffices for many purposes, and is part of how human designers assess manually-authored progressions (Horswill and Foged, 2012).

An observed weakness with manually-authored progressions is that they are sensitive to changes in the underlying data on which they are based: the estimated difficulty values of enemy types, and other design aspects such as frequency of health replenishment or palate cleansers. Constraints on which possible waves may be valid can make progressions time-consuming for a human designer to re-author following a change. These include hard explicit constraints, such as on the numbers of enemies per wave that the engine / game design can support, and also softer desiderata such as the bounds of ideal variance between successive waves. An automated system could allow a human designer to easily re-generate many possible progressions that were known to be valid according to the supplied constraints and then spend time evaluating and exploring these, rather than painstakingly tracking validity themselves during hand-authoring.

By producing a satisfactory system for generating these progressions, we hope to show the applicability of ASP generation to industry-relevant problems in a new domain. An appropriate system should help to solve the issue of reproducing appropriate new combat wave progressions in the presence of changes to player abilities, enemy difficulty ratings or even after the introduction or removal of entire enemy types from consideration – which may all happen during live game development. We also aim to demonstrate the potential of some of the PCG-based design considerations discussed in Sec. 4.4.

## 4.2 The ASP-based approach

We propose an approach involving a domain-agnostic generator system that is able to produce a model of the content based upon a formal set of desired properties produced in concert with designers. As detailed in Chapter 3, ASP allows for a structured generation problem to be explicitly encoded and valid solutions to that encoding to be efficiently found. We present an interrogation of the steps taken to encode the WBC generation problem, along with ASP examples and discussion of the strengths and weaknesses of the approach. In this section we respecify the problem in plain terms, set out our assumptions and signpost the following three sections.

Existing wave-based combat progressions in games are often manually authored and verified,

which makes changes for balance or other reasons labour-intensive. We present a system using a constraint satisfaction approach to generate WBC progression content for a commercial game, according to designer-specified parameters.

Initially an approach to the problem was modelled in a dialect of ASP, HEX (see Sec. 4.5.1), in order to identify relevant information and areas where external computation might be beneficial. Domain-specific information such as enemy types, estimated individual difficulties and observed group sizes were encoded as ASP facts using values drawn from original manual progression designs. Constraints are based upon designer-advised restrictions as detailed in Sec. 4.2.1, observed maximum and minimum wave difficulties, and baseline elements such as forbidding two populations of the same enemy type to be selected for a single wave. In the initial encoding a straightforward combinatorial approach is used to produce all possible valid waves for the specified input, with all combinations that are too large or difficult pruned from generation. As a proof-of-concept, sample constraints are also presented for forbidding or ensuring the co-occurrence of particular enemy types within a single wave.

In the following three sections we cover the sourcing and basic representation of the facts within our domain, an initial approach to using that data to generate combat wave progressions, and our revised approach.

### 4.2.1 Data specification

Existing example data took the form of a complete specification of pre-existing waves designed for the narrative portion of the game (i.e. groups of enemies that would be encountered during the course of a player's progression through the game's normal levels), alongside designer-estimated intended difficulty ratings for each of those encounters

We performed some basic sanitisation to remove encounters where a significant portion of the difficulty arose from external factors (e.g. Boss and Palate Cleanser waves), and then decomposed the encounter difficulty ratings into individual difficulty ratings per type of enemy.

Several initial observations informed the development of the problem coding. In the source data there are 13 enemy archetypes, presented in grouping and combinations of various sizes. The estimated difficulty of groupings of multiple of the same kind of enemy is a linear multiple of the rating of an individual enemy of that kind. The appropriate facts could as easily have been calculated, which would allow for more sophisticated difficulty representations such as awarding bonus multiplier difficulty to larger groups, however at this scale it was equally easy to be explicit. This does expose certain potential weaknesses in the current representation: according to this data, five `warrior`s present the same level of relative difficulty as a single `magic_elite`. While this may broadly be true, it is also clear that the player experience they contribute to in a single wave's composition is clearly different, and it may be appropriate for the system to have ways to reason about that (see Sec. 4.4.2).

Group sizes were drawn from observed instances in the provided data. In some cases these minimum values reflect a design decision: because a group of size 1 of certain weaker enemies that are intended to swarm the player to a degree would add nothing to an individual wave.

In other cases, this reflects elements of the enemy's cosmetic theming: certain enemy types are only ever encountered as pairs for narrative-related reasons, and have combat behaviours that depend on this. Finally, in some cases this may simply represent a gap in the initial data: there is no available group of 6 warriors because no such group was present originally. This is easily rectified if desired, and each of these concepts (where deliberate) could be directly explicitly encoded in the formulation, however one advantage of drawing directly from the original corpus in cases where explicit designer intent may be unavailable is that certain features (such as the original pairing of certain enemies) may be implicitly preserved.

```
% population options; relevant base difficulties harvested from existing data
%% population_option(Type,Count,Difficulty).
population_option(weak_warrior,3,120;weak_warrior,4,160;
                  weak_warrior,6,240;weak_warrior,8,320).
population_option(warrior,2,90;warrior,3,135;warrior,4,180;
                  warrior,5,225;warrior,7,315).
population_option(heavy_warrior,2,140;heavy_warrior,3,210;
                  heavy_warrior,4,280;heavy_warrior,5,350).
population_option(shield_warrior,2,120;shield_warrior,3,180;
                  shield_warrior,4,240).
population_option(axe_warrior,2,180;axe_warrior,4,360).
population_option(ice_imp,3,60;ice_imp,4,80;ice_imp,5,100;ice_imp,6,120).
population_option(fire_imp,3,90;fire_imp,4,120;fire_imp,5,150;fire_imp,6,180).
population_option(flying_warrior,3,225;flying_warrior,4,300).
population_option(beast_pup,8,180).
population_option(beast,2,420).
population_option(maul_elite,1,240).
population_option(magic_elite,1,225).
population_option(axe_elite,1,360).
```

Listing 4.1: Data specification of possible enemy group sizes and relative difficulties.

### 4.2.2 Initial formulation

The initial formulation used multiple facts per wave to specify the relevant groups, which made it difficult to reason over the wave as a whole, and difficulty had to be calculated separately.

Partially as a side-effect of the indirect difficulty representation, the initial formulation was only capable of representing a monotonic increase in difficulty. However, by introducing a lag of three–five waves it was possible to generate progressions that varied in an interesting manner. Unfortunately, in this representation many of the possibilities shot up in difficulty far too quickly until the maximum theoretical difficulty was reached early in the progression, with nowhere further to go and no further variation possible. By intuition: if each successive can only increase in difficulty and the initial wave starts too high or subsequent waves increase in difficulty too far too quickly, many of the possible outputs will reach maximum well before the progression terminates. inefficient representation took a long time to generate long progressions, in part because it could doom itself from the start with an initial high-difficulty wave that left

```
% enemy name projection onto the strings that UE3 expects
eName(weak_warrior,"LesserWarriorActor").
eName(warrior,"WarriorActor").
eName(heavy_warrior,"GreaterWarriorActor").
eName(shield_warrior,"ShieldWarriorActor").
eName(axe_warrior,"AxeWarriorActor").
eName(ice_imp,"HeavyImpActor").
eName(fire_imp,"LightImpActor").
eName(flying_warrior,"WarriorImpActor").
eName(beast_pup,"LesserBeastActor").
eName(beast,"BeastActor").
eName(maul_elite,"TankActor").
eName(magic_elite,"MageActor").
eName(axe_elite,"AxeTankActor").
```

Listing 4.2: Association of the thirteen internal enemy type identifiers with the strings that are expected by the engine integration (see Sec. 4.5.2).

no headroom - a lot of solving time wasted on progressions that could never ultimately work.

### 4.2.3 Revised formulation

Informed by the weaknesses of the initial formulation, we refined the problem encoding to encapsulate each wave as a variable-arity predicate, which simplified reasoning over comparative difficulties.

```
% number of waves
#const n = 100.
#const maxC = 10.
#const pDelta = 30.
#const nDelta = 10.
#const minD = 120.

% generate a fact for each wave
waveNo(N) :- N = 1..n.
```

Listing 4.3: Initial specification of default values for constants, and 'seed' wave facts.

Values specified with the `#const` tag can be overridden by parameters passed to the solver. Here we specify the number of waves, the maximum count of enemies in a single wave, a positive and negative delta for differences in difficulty between successive waves, and an absolute minimum difficulty for any wave. We also make use of the arithmetic range operator (`..`) to generate an initial `waveNo(N).` fact for each wave.

```
group(g(E,C,D)) :- population_option(N,C,D), eName(N,E).
```

L.4.4

Here we perform the name projection based on the translation facts in Listing 4.2, whilst

also wrapping the data within the g/3 predicate labelled by the group/1 predicate — this is a useful formulation that will help us later. Since this projection performs no useful work we could simply reformulate the initial explicit facts to be written in this style instead, which would represent a minor optimisation at the cost of being slightly harder to read.

```
sameType(E,E) :- group(g(E,_,_)).
sameType("HeavyImpActor", "LightImpActor").
sameType("LightImpActor", "HeavyImpActor").
sameType("AxeTankActor", "TankActor").
sameType("TankActor", "AxeTankActor").
```

Listing 4.5: Specification of the reflexive sameType/2 relation, and additional mutual exclusions.

The sameType/2 predicate declares that for each distinct enemy type described within a group fact, that type is the same type as itself — for the purposes of later ensuring that no wave contains two distinct groups of the same type of enemy. This formulation also supports explicitly declaring further pairs of enemy types to be the same type as each other, allowing designers to ensure that they never appear in the same wave together.

```
wave(w(D, e(g(E, C, D)))) :-
       group(g(E, C, D)), D > minD.
```

L.4.6

Listing 4.6 shows the base case wave generation: selection of a single group of a single enemy type and wrapping relevant data within the w/2 predicate, so long as the difficulty of that group is greater than the designer-specified minimum difficulty for a single wave. ASP does not natively support lists of varying length, however by embedding the e/1 predicate within the w/1 predicate, we may later use e/2..4 and still reason over waves interchangeably. Importantly: here the system is not producing specific waves within the progression, simply enumerating valid possible waves according to the provided data. The wave/2 facts describe wave specifications in abstract, which will be selected and assembled into a progression structure by later steps.

```
wave(w(Da+Db, e(g(Ea, Ca, Da), g(Eb, Cb, Db)))) :-
       group(g(Ea, Ca, Da)),
       group(g(Eb, Cb, Db)),
       g(Ea, Ca, Da) > g(Eb, Cb, Db),
       not sameType(Ea, Eb),
       Ca+Cb <= maxC,
       Da+Db > minD.
```

L.4.7

In Listing 4.7 the approach for a wave containing two enemy groups is presented — in this case, we must also ensure that the two groups do not contain the same type of enemy (for our designer-modifiable specification of 'same type' in Listing 4.5), and that the sum count of the enemies within two groups does not exceed the specified maximum (designers may wish to

limit the number of enemies present at the same time for either technical or player-experience reasons). The inequality on line 4 helps to ensure an ordering to our generated waves and ensure perceptual uniqueness: ASP considers two wave specifications containing the same two groups in differing orders to be different waves, and by ensuring that we only generate one of those two possibilities, we avoid later issues.

```
wave(w(D+Dc, e(g(Ea, Ca, Da), g(Eb, Cb, Db), g(Ec, Cc, Dc)))) :-
        wave(w(D, e(g(Ea, Ca, Da), g(Eb, Cb, Db)))),
        group(g(Ec, Cc, Dc)),
        g(Eb, Cb, Db) > g(Ec, Cc, Dc),
        not sameType(Eb, Ec),
        not sameType(Ea, Ec),
        Ca+Cb+Cc <= maxC.
```
L.4.8

For the generation of waves containing three differing types of enemies (Listing 4.8) we make use of a conceptual shortcut by simply adding a new group to any existing wave containing two groups. As before, we ensure that the new group is not of the same type and that the sum of all enemies is not greater than the maximum. We continue to enforce an arbitrary ordering to avoid perceptually identical waves, however in this case we are able to discard the term relating to minimum difficulty as there are no combinations of three enemies that could possibly fall below it.

```
wave(w(D+Dd, e(g(Ea, Ca, Da), g(Eb, Cb, Db), g(Ec, Cc, Dc), g(Ed, Cd, Dd)))):-
        wave(w(D, e(g(Ea, Ca, Da), g(Eb, Cb, Db), g(Ec, Cc, Dc)))),
        group(g(Ed, Cd, Dd)),
        g(Ec, Cc, Dc) > g(Ed, Cd, Dd),
        not sameType(Ea, Ed),
        not sameType(Eb, Ed),
        not sameType(Ec, Ed),
        Ca+Cb+Cc+Cd <= maxC.
```

Listing 4.9: Generation of wave specifications with up to four enemy types (Listings 4.6–4.9).

Finally, for the generation of waves with four different enemy types we re-use the same approach, selecting from the pool of waves that already contain three enemies. One effect of this choice of formulation is that the maximum number of enemy types per wave is implicitly encoded via the presence or absence of individual rules designed specifically to generate waves containing that number of types. Though it is reasonably easy for a user familiar with ASP to simply add more rules that generate additional waves with larger numbers of different enemies (e.g. 5, or maybe even 6), the individual rules become unwieldy due to linear scaling in both the arity of the `e/x` predicate and the number of `sameType` conditions needed. It would further be possible to turn `maxTypes` into an additional `#const` parameter by adding a relevant conditional term to each of the existing rules to make this a designer-controllable concept, however this represents an unnecessary level of additional complexity for the present purpose. Under wildly different generation scenarios where few of the present assumptions hold it could be necessary to develop

an alternate formulation, though as the difference between the formulation presented here and the one in Sec. 4.2.2 indicates, the declarative and comparatively *minimal* representation used in the ASP approach helps to lessen the burden of these changes.

```
1{ waves(waveNum(1), difficulty(D), E) :wave(w(D,E)) }1 :- waveNo(1).
```
L.4.10

In order to begin generation of a progression, a special case rule is presented in Listing 4.10 for `waveNo(1)`. Exactly 1 `waves/3` fact is produced, by selecting among all of the `wave/2` facts produced in the preceding steps. The `waves/3` atoms associate the list of enemies from a particular `wave/2` specification with a position in the progression, and expose the difficulty of that wave for further reasoning. Here we see some of the utility of wrapping the variable-length enemy list within the `e/1..4` predicate, as it means that regardless of the complexity of that specification we can represent it in this rule simply with `E`. Note: though the representation in this section is more efficient than the initial one and so the drawback is less apparent, this approach suffers in theory from the same issue discussed in Sec. 4.2.2 — that many of the possible initial choices are unsatisfactory as they begin with a difficulty value `D` high enough that over the course of the following waves there are either no options or only the same few maximum-difficulty waves repeatedly. A preferable approach would replace this with something like `nextWave(1,minD,minD+pDelta).` and allow the first wave to be generated in the usual way as described by Listing 4.12, however we present this line in its original form for posterity.

```
nextWave(N+1, Dold-nDelta, Dold+pDelta) :-
        waves(waveNum(N), difficulty(Dold), _),
        waveNo(N+1).
```

Listing 4.11: Specification of a range of possible difficulties for the next wave via deltas.

An initial approximation of the trend for increasing wave difficulties is achieved by specifying a sliding window of possible difficulties for the next wave, with a larger possible positive delta than negative (default values are `30` and `10` respectively). Listing 4.11 shows the generation of such a sliding window based upon the difficulty of the preceding wave, dependent on the existence of a further `waveNo/1` fact showing another wave is needed, and entirely overlooking the actual contents of the wave itself. Alternate specifications for `nextWave/3` could integrate historical data from more than one wave, or look ahead in order to lerp towards prespecified future waves (see Sec. 4.3.1) if necessary.

```
1{ waves(waveNum(N), difficulty(D), E) :wave(w(D,E)), Dmin<D, D<Dmax }1 :-
        nextWave(N, Dmin, Dmax).
```

Listing 4.12: Selection of a particular generated wave specification to serve as the next wave.

Exactly one actual next wave is chosen from among the pool of generated wave specifications, according to the difficulty restrictions specified by the associated `nextWave/3` fact. The

deduction of a new `waves/3` fact allows the rule in Listing 4.11 to produce a new `nextWave/3` fact for the following wave, and so on cascading until each `waveNo/1` has been fulfilled by a respective `waves/3`.

```
1{ waves(waveNum(N), difficulty(D), E) :wave(w(D,E)), D <300+5* N }1 :-
    waveNo(N).
```

Listing 4.13: Reformulated wave selection based on progression progress.

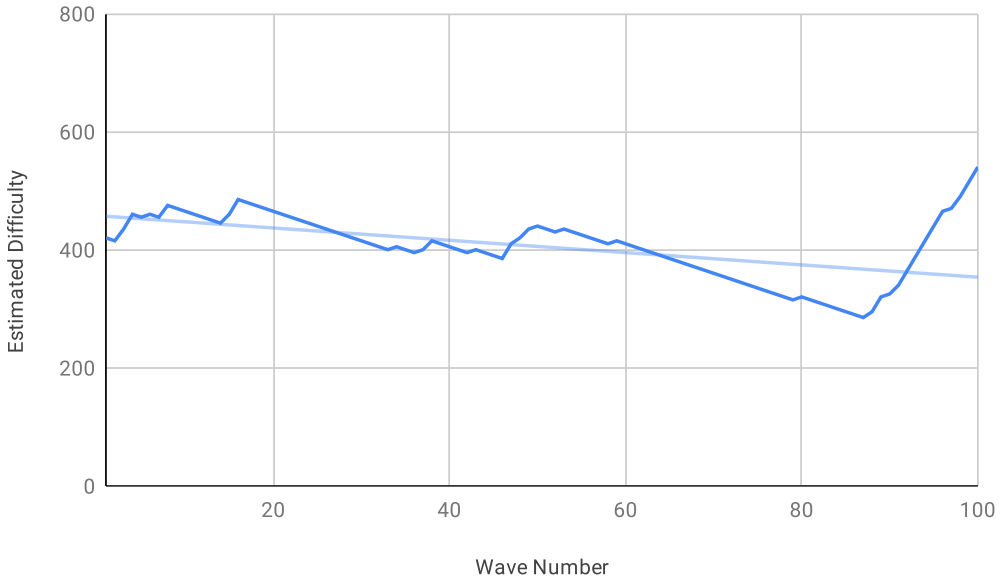



Figure 4-3: An exemplar unintended output from the system demonstrating non-monotonic but eventual increase in difficulty, using the successive window approach in Listings 4.11, 4.12.

## 4.3 Declarative flexibility

The ASP formulation offers a range of new possibilities in comparison to the original hand-crafted approach. It provides the flexibility to easily alter the ground truth of acceptable enemy group size or balancing changes leading to variation in the difficulty ratings, etc, and simply regenerate a new, valid progression. As iteration is quick, the designer can repeatedly regenerate new possible progressions, slowly exploring the space of generative possibilities. Alternatively, minor changes to the formulation can reshape the space of possible outputs. We gather up some of the things mentioned as possibilities in the previous section: designer control over mutual exclusions, variance in approach as part of 'sculpting' desired outputs, potential for explicit declarative domain-relevant constraints.

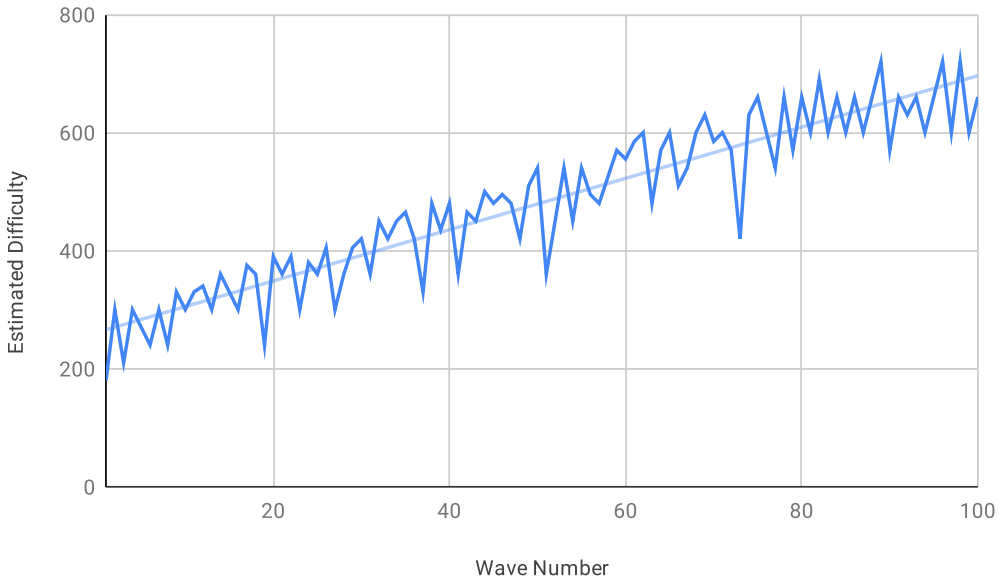



Figure 4-4: A sample output from the reformulated system using progress measure instead of deltas; showing wider variety and more consistent increase in difficulty.

### 4.3.1 Mixed-initiative generation

It is also possible to directly prespecify individual waves if we also suppress the production of a `waves/3` fact for waves that already exist. Listing 4.14 shows a rule to copy supplied prespecified wave elements into a `waves/3` fact, a further modified version of Listing 4.13 that suppresses wave generation if that wave has been prespecified, and a selection of designer-provided 'Boss' wave specifications that match those in Fig. 4-1.

```
waves(N, D, E) :- prespec(N, D, E).

1{ waves(waveNum(N), difficulty(D), E) :wave(w(D,E)), D <300+5* N }1 :-
    waveNo(N), not prespec(waveNum(N), _, _).

prespec(waveNum(20),difficulty(400),e(g("Boss_1",1,400))).
prespec(waveNum(40),difficulty(500),e(g("Boss_2",1,500))).
prespec(waveNum(60),difficulty(600),e(g("Boss_3",1,600))).
prespec(waveNum(80),difficulty(700),e(g("Boss_4",1,700))).
prespec(waveNum(100),difficulty(800),e(g("Boss_5",1,800))).
prespec(waveNum(101),difficulty(800),e(g("Boss_6",1,800))).
```

Listing 4.14: Two routes for producing `waves/3` facts, plus several prespecified waves.

This allows designers to directly specify specific waves as desired, and then use the system to fill in the gaps (c.f. Tanagra, Sec. 2.3). One application of this would be to allow for the specification of known pregenerated 'landmark' palate cleanser and and boss waves within an

otherwise generated progression. As palate cleanser waves in particular show little connection difficulty-wise to the preceding waves in the original data (Fig. 4-2) it is probably acceptable that the existing implementation does not consider the difficulty of future waves when selecting wave candidates, and generates according to the immediately preceding wave only.

```
:- waves(waveNum(N-5), difficulty(Dp), _),
   prespec(waveNum(N), difficulty(Db), _), 2* Dp > Db.
```

Listing 4.15: Difficulty suppression of certain waves prior to prespecified waves.

Listing 4.15 demonstrates a simple implementation of a constraint automatically requiring easier palate cleanser waves five waves before any wave that has been pre-specified (i.e. the Boss waves in Listing 4.14). In this encoding, valid outputs will only contain palate cleansers that are less than half as hard as bosses they precede. More sophisticated formulations are possible — one weakness of the listed approach is if designers hand-specify any two waves that are five waves apart and do not obey this constraint, the whole problem will become unsatisfiable, with potentially little useful feedback as to why. More sophisticated ASP debugging tools could help alleviate this problem (as in Brain, Cliffe and De Vos (2009)), however in this instance it is enough to add a clause excluding the existence of a `prespec(waveNum(N-5), _, _)` fact.

However, if a smoother progression difficulty curve were desired, the `nextWave/3` rule could be replaced with two variants: one that operates as presently in the absence of future information, and another that aims to 'lerp' towards known future waves.

Another possibility relates to the potential to regenerate only parts of a progression. The ease of iteration across differing outputs from the generator facilitates casual exploration of the space, however the system as presently described only allows for complete regeneration of the whole progression each time — there is no way for an interested designer to preserve desirable portions of an individual progression and 'roll the dice again' on the remainder. However, by passing the generated facts relating to the desirable waves back into a new run of the problem as prespecified waves akin to the previous landmark waves, a new progression can be obtained that differs only over the previously-undesired waves.

### 4.3.2 Further constraints

Another potential avenue for investigation would be tools to allow designer interaction with the generation process. This could initially involve allowing additional designer-provided constraints like the ones available to players in Compton, Smith and Mateas (2012) – 'enemyX must be introduced before enemyY', 'enemyW must never appear in two waves consecutively' or 'enemyA may only appear if enemyB is present', etc. — all of which are reasonably easily represented in the formulation. Further options could include designer specification of a more sophisticated explicit tension curve to be matched during wave generation, to require that the progression of difficulty from one wave to the next is as close as possible to a desired curve. Finally, a custom interface would be needed to support direct designer manipulation of gener-

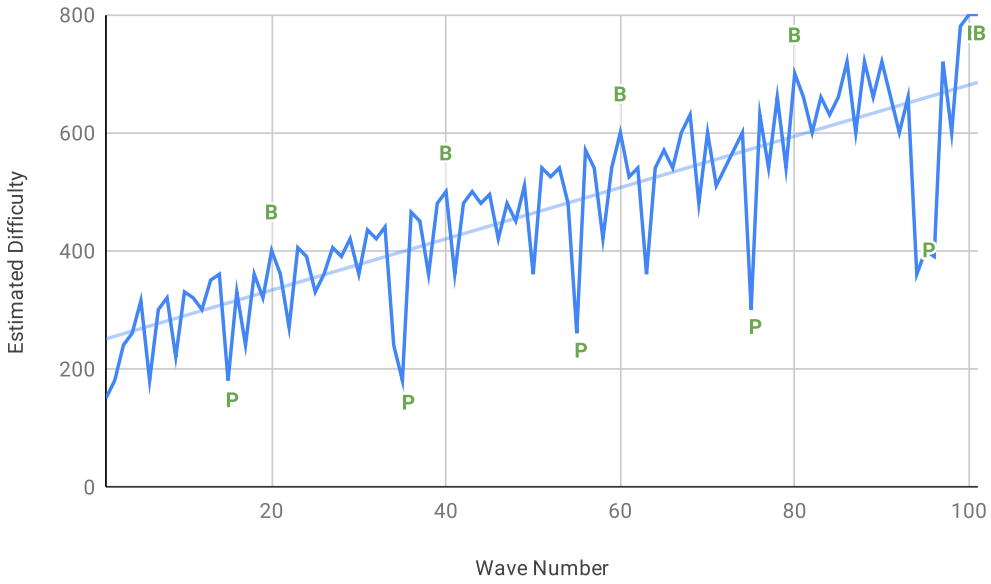



Figure 4-5: A sample output showing prespecified labelled landmark waves: 'Boss' waves with asserted difficulties, and 'palate cleanser' waves with difficulty restrictions.

ated progressions, allowing addition, removal or transposition of waves, or changes to individual wave compositions. Potentially, specific edits or waves could be marked for preservation and the rest of the progression re-generated in a manner consistent with the preserved elements, as in Smith, Whitehead and Mateas (2011).

## 4.4 Design implications

The system as described so far is a tool to assist designers in producing a progression for wave-based combat, and as such is intended to save the labour that would otherwise be spent generating (and then potentially as-necessary re-generating) such a progression manually. To achieve direct comparability with the current approach, designers would use the system once, verify that the generated progression was acceptable, and then include it in the game to be shipped — i.e., in this scenario the system is used only at 'design time'. However, once we have an automated 'designer-in-a-box' several new possibilities for alternative designs become apparent. If the designer is confident that the problem is sufficiently well formulated that any output from the generator is acceptable, then the generator may be used to produce a new progression each time a player desires, ensuring that their experience is less likely to fall into rote predictability than if the the progression were static and unchanging — without additional input, the system could also be used at 'load time'. Though the general claim that PCG provides 'increased replayability' is rightly criticised, for this particular application interested players are expected to replay the challenge frequently in an attempt at mastery regardless —

the application of PCG to this problem may simply help to keep the experience fresh, and reduce the probability of players experiencing undesirable or overwhelming levels of frustration as a result of becoming 'stuck' on a particular wave (Boulton et al., 2017; Smith, 2014).

The system as described is currently only capable of reasoning over available data and producing waves based solely on previous wave difficulties (or, with minor potential modifications, also lerping towards future boss or palate cleanser wave difficulties). However there is also potential to use the system at 'run time', to generate new waves in a just-in-time fashion by reasoning over previous waves re-entered as prespecified data, and additionally also over data gathered from the player's interaction with the prior wave.

### 4.4.1 Dynamic generation

There are two potential extensions to the work that would necessitate and lead to further development of the system itself — both are forms of mixed-initiative work as described in Sec. 2.3.

'Responsive' wave progression generation would be a form of player-driven PCG, in this case similar to dynamic difficulty adjustment systems. If the wave selection system runs during gameplay, it can use information about the player as input to the reasoning in order to produce a wave progression that is in some sense tailored to the player. At the most basic level a simple player model might include variables such as $\Delta$`health`, in order to track the health that a player lost or gained during a single wave as an approximation of how challenging they found it. In cases where expected $\Delta$`health` differed from actual $\Delta$`health`, the system could use this as a cue to adjust the difficulty of upcoming waves. More sophisticated implementations could track the weapons or abilities that players choose to use, as in Hastings, Guha and Stanley (2009), in order to generate further content that provides satisfying opportunities for those skills. This risks forming a feedback loop that converges on an overly small number of perceived-well-suited wave compositions and so a deliberately erratic wave could be generated to encourage diversifying the player's skill set, but only as a design-guided decision.

Rather than pre-generating the entire progression at level load time, a modification of the existing ASP encoding could be used to select each successive wave on-the-fly. This would allow runtime metrics relating to the current play session to be used as an input to the reasoning process, potentially allowing each next wave to be responsive to the player's recent or overall performance. Even a simple model consisting only of the time a player took to clear the previous wave and the amount of damage they took in doing so could be used to tailor the experience to individual players' ability levels.

In addition to the underlying problem encoding, ASP fragments representing the player's progress and performance in the level so far could be composed to influence the generation of the next wave. It would also be possible to include special designer-provided constraints that are only active under particular conditions, in order to inject a greater degree of variance into the possible outputs from the system — either to reward particularly successful players, or provide support or additional training opportunities for players that need it. In this dynamic

context the speed of the system becomes more relevant, as the solving process for generating an answer set representing the next wave must occur between each wave, and so cannot be hidden in the level load times or pre-generated during development.

### 4.4.2 Further work: skills-acquisition model

The existing work produces a progression of successive waves composed of mixtures of enemies, but initially implicitly assumes that the player character has acess to all abilities, and that the player is equally proficient with all of them. If the generator is operating at runtime, it should be additionally possible to reason over a simplified model of player experience with given abilities and enemies such that the player is engaged by constant introduction of new concepts; Butler et al. (2015) present an application of this concept using ASP for an educational 2D puzzle game, *Infinite Refraction* (see Sec. 3.3.1). As presented, both pre-generated and dynamic wave generation use ASP primarily to quickly enumerate and then constrain large combinatorial problems — the declarative nature of the approach makes this comparatively straightforward — but fundamentally the problem could still be solved using other approaches. One benefit of using ASP however is that it further supports reasoning over increasingly complex and highly constrained problems.

As noted in Sec. 4.4, many of the interesting constraints in this domain relate to the overall complete progression of waves, and specifically the order in which abilities, weapons and enemies are introduced. Correct modelling and generation of valid progressions in this manner will require annotation of each wave in order to track individual concept introductions, as described in Butler et al. (2015). Full progression generation will necessitate designer input on appropriate weights for each concept, as well as indication of which pairwise concepts warrant additional weight – though there is scope for this information to be inferred from existing content.

The challenge mode feature in many games is only unlocked once a player has progressed through the entire single-player storyline campaign and unlocked all of the different weapons and/or essential character abilities: as a consequence, players begin the wave-based combat with all weapons and essential abilities available, and the assumption on the part of the designer that they understand how to use them. This means that the nature of constraints on a particular wave relate entirely to its composition, and not to which abilities or weapons the player may have, as it is assumed that all are available.

In contrast, the main narrative storyline portion of action/combat games typically introduces new weapons and enemy types in a gradual succession in order to allow the player to develop mastery over each in turn. As a concrete example, enemies with a shield are particularly difficult to defeat without a heavy weapon such as the axe. An approach used in some games would be to present the introduction of a new enemy kind immediately *before* the introduction of a new weapon — in order to emphasise to the player the difficulty of defeating that enemy type, and then contrast the ease with which it could be fought with the new weapon once it has been acquired. There would then be an acclimatisation period, where through the next segment of narratively-connected waves the player would be presented with increasingly

difficult combinations of enemies including the new kind.

This approach is similar to the 'skill atoms' concept and mastery process described by Deterding (2013), where the player's skill with the new weapon and the player's understanding of the new enemy kind are separate, but linked feedback loops that are clearly introduced and built upon as the player progresses. New movement abilities and their paired environmental obstacles are introduced in a manner analogous to weapons and enemies, and a variety of each are often introduced throughout a game's main story mode in order to maintain interest and develop complexity. Hence, there is frequently a designer-developed 'progression' of new concept introductions embedded within the storyline portion of any given game, whilst in contrast arena modes typically introduce no new concepts, simply new combinations of enemies in a focused experience, unlike that found within the narrative.

Using an appropriately formulated encoding of the available concepts and their relative inter-dependencies, an alternative wave-based-combat-style progression system could be generated that is tailored towards introducing new players to each of the gameplay concepts in a reasonable (novel) order, without the associated narrative. A possible approach to this could be modelled on the one presented by Butler et al. (2015), which produces a problem progression from basic to complex puzzles by maintaining a player model of which concepts the player has mastered (successfully completed one or more times). In this context each weapon or enemy is a concept, but also co-occurrences of pairs (or even possibly triples, etc.) of skills are also concepts. A 'problem' (combat encounter/round) with fewer concepts is 'conceptually simpler' than one with many concepts. A problem where most concepts have been previously mastered is also simpler than a problem with mostly novel concepts. This allows a partial ordering to be produced which may be updated as the player masters new concepts, and used to select appropriate next wave or concept introductions. While this feature is only appropriate for new players and so is not repeatedly replayable in the same manner as more traditional arena approaches, it is also a necessary precursor to whole-game generation in this genre Togelius et al. (2013a).

In order to generate a satisfying and challenging progression of wave compositions, some formal model for the approximate difficulty of a wave will be needed. Butler et al. (2015) describe a system for generation a progression of successively more challenging graph-based puzzles, where each puzzle is annotated with the concepts needed for its solution. Concepts include individual puzzle pieces, combinations of puzzle pieces and certain domain-specific ideas. Each concept is provided with a designer-specified difficulty 'weight' (though many of these are automatically populated), and novel concepts that have not appeared in any puzzle in the progression so far receive a bonus. Whenever a new puzzle is needed for the progression, all of the novelty bonuses are recalculated and weights summed, and the puzzle with weight closest to an empirically chosen target value is used. For each puzzle that a player successfully completes, concepts in that puzzle thereafter receive a small penalty to their weights, meaning that initially concepts are highly weighted due to unfamiliarity, but as the player becomes more practised, difficulties become discounted. Using this approach, players are slowly introduced to new concepts individually, as the bonus system discourages introduction of multiple new concepts at once.

A similar approach may be appropriate for combat wave progression: each additional enemy adds to the wave's difficulty according to the challenge of the enemy type, with further difficulty based on the co-occurrence of specific enemy pairs, and bonuses for the introduction of new concepts. An automated combat wave progression system should therefore be capable of generating all valid waves under these restrictions, annotating them with weights for each concept, and then selecting a suitable progression of waves to return as output.

## 4.5 Practical considerations

In this section we discuss aspects of the system unrelated to the actual ASP encoding or the design possibilities that the designer-in-a-box paradigm surfaces. Specifically, we consider initial comparisons between two varieties of ASP solver that were available during the project, discuss the factors that led us to choose one over the other, and provide a brief overview of the engineering issues encountered while integrating this system with an existing game engine.

### 4.5.1 dlvhex comparison

Original implementation efforts (4.2.2) focused on `dlvhex` and an intended system architecture that would rely on use of external atoms to provide e.g. Gaussian distributions, links to OWL datastores (see Sec. 7.2.3) and links to a game process for dynamic generation. Several of these elements turned out to be infeasible or not sufficiently performant.

The `dlvhex` system integrates clasp and gringo but does not allow for direct use of Clingo for pure ASP code, so the problem was modelled in HEX to take advantage of external computation and OWL knowledge representation.

`dlvhex` does not support the full ASP language standard – in particular, it does not allow the `min {options} max` 'choice rule' syntactic sugar. It is possible to approximate behaviour similar to a choice rule using a disjunctive head (`option` $\vee$ `-option`, where '`-`' represents strong negation) and a `#count{...}` aggregate, however this approach is comparatively inefficient due the the greatly increased size of the grounded representation. Efficient HEX-programs must therefore be written in a manner that minimises the use of choice rules. The availability of plugins for non-ASP computation means that other problem formulations are possible: in the HEX implementation, first up to three enemy types are chosen, and for each of these several integer samples taken from a type-specific population size distribution, using an external atom. These samples are then combined into all of the allowable waves under the specification given above.

One of the relevant primary advantages of using existing technologies such as OWL and ASP is the availability of pre-built tools and libraries, and sound theoretical backing in existing academic research. By building on existing code and implementations, much effort may potentially be saved; however, often systems have been developed for other purposes and contexts and so may only partially provide needed functionality or present idiosyncratic restrictions or other complexities, which are often not fully documented.

Early comparative tests of simple wave generation using both Clingo and `dlvhex` revealed a significant speed difference between the two systems. In order to allow later comparative evaluation of the system, development of a new integration with the UE3 engine became necessary — and as a number of features supported by newer releases of Clingo (such as Python integration and multi-shot reasoning) appeared to allow functionality similar to many of the original reasons for using `dlvhex`, the speed increase alone indicated that Clingo may be a more appropriate choice of ASP solver. Aside from improved execution times, the use of Clingo directly provided a number of other benefits. As the full ASP language standard is supported, problems could be modelled more naturally and efficiently using constructs such as the choice rule, native aggregates such as `#min` or `#max`, and external `#constant` declarations. Multi-shot reasoning is a recent feature that allows for the evolution of an ASP model over time, in response to additional external data. Instead of discarding current grounded models and partial solutions in order to restart reasoning from scratch, newer versions of Clingo support reactive updates for efficient reasoning. There is also a large and active development community around the Clingo tools, and a number of related projects that seemed potentially useful — e.g. 'Xorro' is an external preprocessor that allows addition of specially crafted constraints to an ASP problem, in order to retrieve a random representative sample of all answer sets without requiring exhaustive computation.

The system and timings as presented Table. 4.2 is a snapshot of the work as it progressed towards the implementation as described in Sec. 4.2; in this test there is no consideration of difficulty or same-type reasoning, the formulations were each a simple enumeration of all valid wave descriptions. Up to 13 enemy types are potentially specified, with up to 3 types chosen for each wave. Consideration of additional types increases reasoning time taken for the problem.

| enemy | Clingo | | `dlvhex` | |
|---|---|---|---|---|
| variants | time | waves | time | waves |
| 3 | 0.16 s | 100 | 14.26 s | ~101 |
| 4 | 0.20 s | 250 | 49.76 s | ~277 |
| 5 | 0.32 s | 509 | 204.32 s | ~562 |
| 6 | 0.40 s | 770 | 1351.50 s | ~1142 |
| 7 | 0.68 s | 1131 | * | - |
| 8 | 1.08 s | 1549 | * | - |
| 9 | 1.06 s | 1575 | * | - |
| 10 | 1.36 s | 1876 | * | - |
| 11 | 1.60 s | 2244 | * | - |
| 12 | 1.94 s | 2644 | * | - |
| 13 | 2.78 s | 3060 | * | - |

Table 4.2: Comparison of Clingo and `dlvhex` timings on similar wave generation problems. Tests performed on a Windows 7 desktop with a 2.8GHz quad-core Intel Core i7-930 processor and 12GB RAM.
*: >1500 s

These results indicate a distinct scaling problem with `dlvhex` enumeration speed — making it infeasible to consider many kinds of enemies, or complex restrictions on difficulty.

Table. 4.2 shows the execution times of a number of wage generation tests performed using the standalone versions of Clingo and `dlvhex`. These times represent simply generating all of the waves valid under the respective program definitions, without consideration for additional constraint relating to individual difficulty or variety, and without the translation harness that converts an answer set into a usable wave specification. The number of enemy types considered affects the grounded size and therefore the execution time of a problem. Each configuration was run 5 times, and the results averaged. Problem definitions differ, which explains the difference in waves generated. For Clingo, the number of waves found for each configuration was consistent on each run: the complete enumeration of all possible options, without any waves that were overall too easy, too hard or too large. For `dlvhex` the number of found waves varied, due to the use of random Gaussian distribution generation as part of the problem definition. It should be noted that these results are not intended as a rigorous empirical evaluation of the two approaches, merely as an intermediate result revealing a significant difference in execution times between systems.

A range of considerations were relevant to the choice to discontinue further development of the `dlvhex`-based approach for this work. As implemented, it is capable of loading information from a basic OWL ontology, but still relied on monolithic, domain-tailored HEX/ASP code for actual generation. The significant execution times of even comparatively simple HEX-programs placed a restriction on the complexity of constraints that could be reasoned over. Even simple approximations of difficulty thresholds as considered in the Clingo comparison approach considerably increased the time taken to find a usable result. The version of the problem presented in Table. 4.2 (simple combat wave enumeration) has a reasonably straightforward combinatorial output, for which ASP is useful but not necessary. The fuller model of the more complex potential restrictions on wave progression generation in Sec. 4.2 (plus any future considerations such as unlockable skill or weapon progressions, enemies that may only be fought with certain weapons, further mixed-initiative or designer-guided constraints) is made possible via the comparative speed of the Clingo implementation.

### 4.5.2 In-engine implementation

In order to demonstrate run-time generation of content via ASP, we developed an approach for generating varied and challenging successions of combat waves for commercial third-person action/combat games, within Unreal Engine 3 (UE3). This approach used the faster Clingo solver over ASP modelling the combinations of enemies with differing quantities and strengths, in order to design for a space of possible experiences and allow quick re-generation after balance changes at design time.

Comparable to manual specification, 'progressions' of waves can be pre-generated that follow a trend of increasing difficulty, with variance, analogous to the puzzle progressions presented by Butler et. al. Butler et al. (2013, 2015). We present the current work as an intermediate

step towards a system for multi-purpose content generation within a commercial game engine.

In order to support ASP-solving during either design or at play time, it was necessary to integrate an ASP solver and grounder with the game editor and engine. Early work investigated the applicability of `dlvhex` (Eiter et al., 2006), which offered support for integration of external computation, however speed comparisons and recent features supported by Clingo (Gebser et al., 2014) led to that solver being more applicable.

A number of 'off-the-shelf' solvers are under active development in academia for reasoning over ASP problems[2,3]; one of the advantages of this approach is that the hard algorithmic work can be performed by a highly-optimised library developed elsewhere and can benefit from future updates. To generate content for use in a game at runtime or design-time, it is necessary to integrate a solver with the game engine and/or editor — for the purposes of this research, this has been Unreal Engine 3 and 4 (UE3/UE4)[4].

As a first step to augment traditional designer hand-authoring of combat wave progressions, we present a design space approach to using ASP to generate combat progressions. Individual waves are generated according to the original designer-provided parameters detailed previously, and consist of some particular composition of enemy types and numbers, associated with a total estimated difficulty for the wave based on designer-specified approximate difficulties for individual enemies — notably, this does not account for complementary interactions between differing enemy types. Waves are composed into a progression of suitable length according to a small number of constraints on variety and maximum difficulty deltas between rounds, and a randomly selected answer set describing the entire wave progression is emitted as JavaScript Object Notation (JSON) by the Clingo plugin during the loading process of a challenge mode game level (Smith, Padget and Vidler, 2016). This can be easily converted into the internal wave progression format and used to spawn the enemies described for each successive round.

These pre-generated wave progressions are comparable to designer provided ones in that they are fixed from the start of a specific run of a challenge mode level, however they can vary between individual runs of each level. This may help to avoid the challenge becoming 'stale' as players memorise the precise configuration of enemies in each wave — more importantly, the wave generation process can be responsive to balance changes in the abilities and/or strengths of the enemy characters.

### 4.5.3 Progression analysis

It is important to ensure that the system produces combat wave progressions of acceptable quality — casual evaluation by specimen observation and comparison to existing artefacts is sufficient to guide the development and evolution of the formulation, but thorough coverage and acceptance testing is also needed, and requires appropriate methods. As detailed in Sec. 2.5 there are a range of potential approaches available for evaluating the output, of which computational analysis and two forms of human evaluation are likely to be most relevant.

[2]Clingo: `https://potassco.org/clingo` — accessed 27 November 2020
[3]`dlvhex`: `http://www.kr.tuwien.ac.at/research/systems/dlvhex` — accessed 27 November 2020
[4]`https://www.unrealengine.com/en-US/` — accessed 27 November 2020

The simplest method is the use of computational metrics, as described in Chapter 6 — sampling the output of the system and using appropriate calculations to investigate salient properties. Many of the metrics covered in existing literature are likely to be inapplicable for this comparatively novel domain, though the pairwise metric *compression distance* (Shaker et al., 2012) may help to indicate the presence or lack of meaningful variance in the output space. There are also a range of potentially appropriate additional metrics — as an initial proposal: the average absolute difference in difficulty between successive waves — or perhaps the variance of that measure — could indicate the 'smoothness' of a progression. Though there are only two reference datapoints it would also be possible to perform direct comparative analysis between generated progressions and the existing designer-produced progressions.

Further qualitative evaluation of the system and sampled output could be available via consultation with professional game designers; domain experts who may be able to provide feedback on the performance of the system and quality of the generated progressions. They may be able to provide a critical evaluation of how a given generated progression compares to a manually produced equivalent.

Finally it would be instructive to run a side-by-side user study comparison of both generated progressions and the existing human-designed setups, in order to assess the perceived quality of the system's output from end-users' perspectives. If possible, it would also be useful to attempt to assess the quality of a full progression designed to teach a player the game mechanics from scratch by introducing weapons, enemies and abilities individually and providing safe opportunity to practise with each – this would require a range of additional constraints on generation and be potentially difficult to produce without ASP. Comparative analysis would not be possible in this case as existing progressions assume the player is already familiar with all weapons and abilities, and so they are available from the start.

## 4.6 Discussion and further possibilities

In this chapter we have described the integration of ASP with a commercial engine and the generation of content in the form of combat wave progressions. This algorithmic approach potentially offers benefits over the traditional trial and evaluation techniques, as it allows designers to think in terms of a population of possible valid wave progressions, and quickly re-generate new progressions in response to design changes. Further, we detailed ways in which this work may be extended in order to make better use of the possibility afforded by ASP to generate complex constraint-driven outputs, and potentially provide a more responsive experience for the player. Finally, we sketched three ways in which this work may be more thoroughly assessed, in order to provide a point of reference for future academic work in this area.

The current combat wave generation system is intended as a development project, as a proof of concept in applying the ASP-based design space sculpting approach to a problem with industrially-relevant scope. Aside from similarities to Butler et al. (2015) there is no clear existing academic attempt to generate content for this domain, however there is an opportunity for comparative evaluation with the existing manually-produced instances, and expert analysis

by the creators of that implementation. Further development in this area could focus on developing an ASP-backed Domain-Specific Language (DSL) containing concepts relevant to skills-based progression generation and the various semantic constructs described in Sec. 4.3.2. It should be possible to annotate existing objects and compose or select design constraints with as little friction as possible in order to encourage experimentation and rapid iteration. Development of a decoupled generator system as described would further approach the goal of being able to produce 'plug-and-play' PCG middleware for a wider range of games and content types (Togelius et al., 2013a).

The specific work described in this chapter is intended as an initial step towards the goal of general content generation as detailed by Togelius et al. (2013a). It represents an integration of an ASP solver with a commercial engine, and use of an ASP problem encoding to generate combat wave specifications. Under the current implementation, generation is performed by a handwritten ASP problem encoding detailing the specific constraints that define the problem space, along with parameters specified by designers. These parameters represent both engine limitations relating to the number of on-screen characters that can be reasonably be supported in a single wave, and design decisions about the number of different types of enemy that a player can comfortably simultaneously consider during combat. During original development of the wave-based combat system for this game, these parameters were checked by a validation script that ensured they were not violated by a human designer's wave assignment — with ASP, these constraints become part of the generation process, and so do not necessitate a separate verification phase.

An important minor extension to the existing implementation would be to increase the sophistication of the wave difficulty evaluation heuristic, by adding modifiers to represent the co-existence in a wave of particular pairs of complementary enemies. These could be specified by designers from expert domain knowledge, and applied to relevant classes of enemies based on shared properties such as the ability to fly or use ranged attacks. Another potential feature would be support for variable-length progressions, by allowing the player to specify a desired number of rounds and then generating a balanced progression appropriately. Other more substantial extensions include the possibility for dynamic generation or more complex player modelling for a tutorial progression, as detailed in Sec. 4.4.2.

# Chapter 5

# Constrained Dungeon Design

In this chapter we build on the Answer Set Programming (ASP)-driven content generation approach detailed in the previous chapter and present a more complex application of ASP to content generation. We begin by introducing the process of level design and the concept of level greyboxing. We introduce and define the concept of dungeons in games, and link it to the greybox generation problem; covering the core structural features, existing generators in literature and a particular useful generation pattern. We detail an approach that has been developed for transcribing the structure of dungeons in existing games, and note that the level of abstraction it provides may also present a useful opportunity for top-down generation of dungeon structure. In Sec. 5.2 we specify the problem we intend to tackle, briefly outline an initial attempt and the issues it uncovered, and present in detail our approach for dungeon graph generation with some discussion of the good and bad outputs. Then we discuss the implementation of a plugin for an industry standard game development tool to facilitate exploration of generated greybox levels using our approach. Finally we review the capabilities of the system as presented, and lay out some possible directions for further development work.

Typically, development of a level of an action adventure game will involve a 'greybox' phase, where the basic flow of the level has been laid out, including key landmarks, challenge areas and sufficient connective geometry to result in a playable level, but detail is not specified and all geometry is represented by untextured (grey) cuboids. This allows for easy iteration to ensure that pacing and feel of the level are satisfactory before investing time in adding art and other details, as later changes would incur greater costs. The greybox phase of development requires that a level designer must manually evaluate the validity of the design according to whether it is possible for a player to actually complete it, and how well it satisfies any given design requirements. They may maintain a mental representation of their intended level structure, and this may also be abstractly diagrammed within design documents but it is rarely explicitly described within the development tools, where basic geometry without explicit semantic association may be used to represent each area.

In the early stages of level design, a range of important considerations are relevant, from strict playability constraints such as ensuring that a valid route exists from the start to the

end of a level and keys are available before the doors they unlock, to 'softer' constraints such as ensuring key locations are visible from certain areas. Many of these considerations are difficult to enforce or even express in traditional editors such as Unreal Engine 4 (UE4), so these constraints must be understood, remembered and self-enforced by designers.

By explicitly representing and reasoning over these constraints, an integrated ASP-based generator can provide initial content that is usable and correct, provided that the constraints have been specified correctly (Smith and Mateas, 2011; Smith et al., 2012). To limit the design space to tractable sizes, a multi-part approach is taken. An abstract model of the final content is generated first — in this case, a level flow sequence specifying the kind and order of challenges to be encountered, analogous to the progressions in the combat wave case. Unlike the wave progression detailed in Chapter 4, this is not a linear sequence of challenges but a graph representing physically-connected concepts, with additional edges recording necessary temporal relationships. That graph is then laid out by a non-ASP module and instantiated as a basic playable space within the editor, though there is scope for more detailed post-processing or further refinement across additional modules.

In action-adventure games, players typically have access to a range of possible abilities which they can use both to move around the game world and to interact with other entities within it — either passive entities which can be moved or activated or talked to in order to solve puzzles, or active, hostile entities which must be fought and defeated. Progression through a game level is closely linked to correct use of these abilities in order to complete local challenges, though often multiple possible routes to correct completion of an individual challenge are possible. Broadly, challenges fall into one of three categories: traversal (correct use of movement abilities), combat (use of offensive/defensive abilities to defeat one or more enemies) and puzzle challenges (use of reasoning or trial-and-error to bring game objects into a correct configuration), though elements of each often overlap, and other categorisations are possible. Often, these challenges will occur within game areas specifically designed to support the relevant activities and completion of the challenge within an area will allow the player to move on to one or more successive areas, though occasionally levels may be designed to ensure that players must return to previously-visited areas and find the next challenge there. In addition, progression is often gated by situations that require the acquisition of a particular item or ability elsewhere in the same level, or in another level. These combinations of situation and item/ability are often termed 'lock and key' puzzles, regardless of the nature of the individual items or abilities (Dormans, 2017; Shaker et al., 2016).

In this chapter we introduce the vocabulary for several important dungeon generation concepts; our extensions to an existing method of visualising dungeons' spatiotemporal structure, and several examples of related work that have informed the current approach. We then detail our initial method for generating content within this space, and the refinements we performed, along with some sample outputs. We describe how this was integrated with a commercially-relevant suite of editor tools for games, and suggest a number of promising directions for further work.

## 5.1 Dungeons in games

'Dungeons' are a particular kind of contained episodic experience and associated playable area within many action-adventure games. Characterised by a degree of detachment from the main plot of the game (if any) and a complex physical layout, their self-contained nature, somewhat formulaic structures and typical lack of direct verisimilitude to any real-world space make them well-suited as candidates for procedural generation. In this section we define a number of aspects of typical dungeon implementations that are relevant to our generation concerns, as well as a number of common aspects that we place beyond the scope of the current project. We note the similarities between dungeons and the broader concept of 'levels' used in many modern 3D action-adventure games, and highlight the differences that make the dungeon generation problem more tractable. In Sec. 5.1.1, we provide additional detail on 'lock and key' puzzles — the chief structural concern relevant for generating complex dungeon layouts — and define the difference between *local* and *non-local* challenges. Sec. 5.1.2 covers a number of existing dungeon generators described in literature, with particular attention to how they approach the challenge of generating consistent layouts. Then Sec. 5.1.3 describes a common generation pattern that informs the approach described throughout the rest of this chapter.

> "The appeal of a Zelda dungeon is in its intricacy. A good dungeon starts off feeling overwhelming, full of buttons and doors and strange obstacles. As you make progress, solving puzzles and acquiring items, it all slowly begins to make sense. You get a complete understanding of how the passages flow and the rooms all weave together, as if you've just solved a tough math problem or learned the secret behind a magician's trick."
>
> Jason Schreier, *Kotaku* (2019)

In their survey of design patterns in dungeons in Role-Playing Games (RPGs), Dahlskog et al. describe dungeons as "levels with a spatial puzzle quality", and accept the 1983 *Dungeons & Dragons* description ("A Dungeon is a group of rooms and corridors in which monsters and treasures can be found.") as their baseline (Dahlskog, Björk and Togelius, 2015; Gygax, Arneson and Mentzer, 1983). *The Legend of Zelda* series of video games frequently contain multiple dungeons with varied designs, and introduced many of the design elements that have become staples of the genre. Common features of dungeons across multiple games include:

1. Increasing access: some of the dungeon is initially inaccessible to the player until they have overcome various obstacles such as combat or puzzles, discovered routes that may be hidden or require a degree of player skill to traverse, or located and used keys (or keylike items) on locks (see Sec. 5.1.1).

2. A variety of obstacles (e.g. combat, puzzles), to avoid repetition.

3. An element of player choice or exploration, rather than a highly linear sequence of events as in e.g. platform levels (van der Linden, Lopes and Bidarra, 2014).

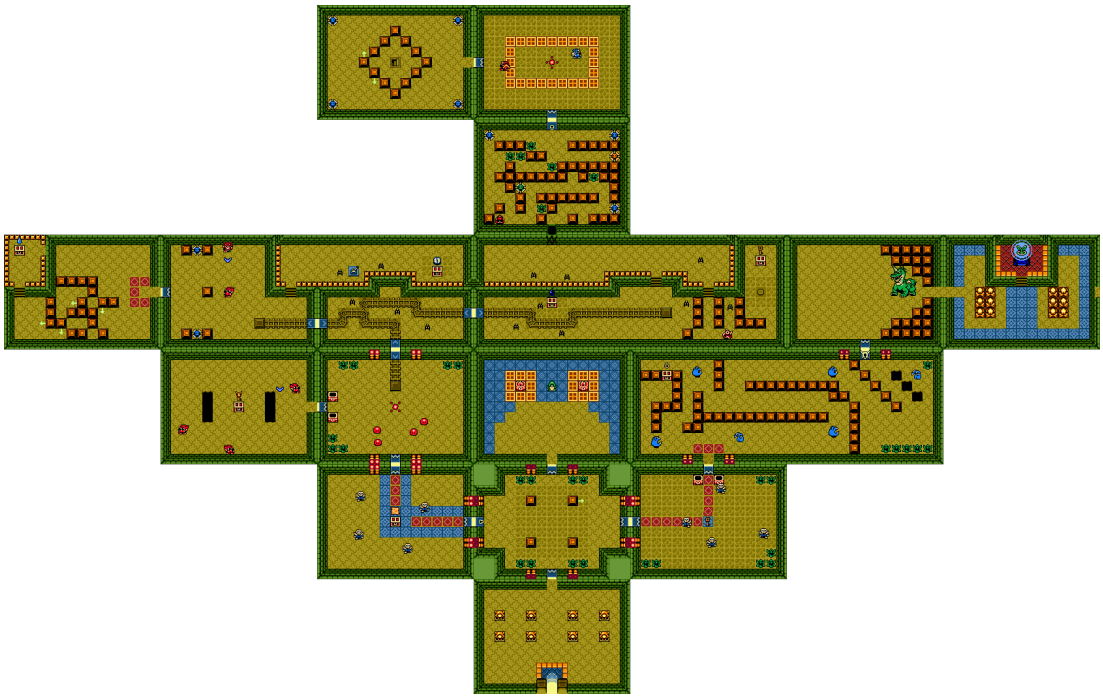


Figure 5-1: An example layout of 'Gnarled Root Dungeon', the abstractly eagle-shaped first dungeon the player encounters in *The Legend of Zelda: Oracle of Seasons* (Nintendo, 2001).

4. Optional or hidden routes or rewards that make progress through the dungeon easier but are not critical for completion.

5. Some degree of 'backtracking' or return through previously explored areas: a common pattern is to have the player encounter one or more locked doors early in the dungeon, to indicate that they should search for the relevant key(s) and then return (Dormans, 2017).

6. A 'Boss': a tougher-than-usual combat or scripted encounter that serves as the final obstacle and goal within the dungeon, and whose defeat typically advances some story-related purpose (Heijne and Bakkes, 2017).

In Sec. 5.2, we provide an approach to generate specifications for dungeons that display the above structural features, while abstracting over the particular choices of 'Boss', rewards or obstacles.

Some of the features described above are visible in Fig. 5-1, which is a map of the nineteen rooms of a dungeon in an early 2D action-adventure game (Nintendo, 2001). Certain elements, such as variety in environments or room layouts and the potential for exploration, are clear. Some of the most important information — the structure of a players' expected progress through the level, including backtracking and choice points — is particularly difficult to read from this presentation of the dungeon, which motivates the graph abstraction approach described in Sec. 5.1.4.

Fig. 5-1 also shows certain features that have become established as repeated design tropes of dungeons in many games (including *Zelda* games), but are not necessarily core elements of the dungeon concept. These include a loose symmetry to the overall layout of the dungeon, and also a recognisable shape to the dungeon's outline, which in this case is intended to resemble an eagle. Other elements of some dungeons include characteristic mechanics with a broad-ranging effect on the accessibility of various areas within the dungeon, such as the ability to alter water levels to flood or uncover regions, or the possibility of revisiting the same dungeon during differing time periods to observe the forward effects of certain actions. As these concerns are either largely cosmetic or require significant bespoke alterations to the normal dungeon structure, we consider them out of scope of the current project. Sec. 5.2.1 provides more detail on the specific aspects our system supports.

Though the examples given so far have been framed in terms of 'rooms and corridors' as in the original 1983 definition, dungeons are not inherently limited to 2D or grid-based representations. Dahlskog, Björk and Togelius (2015) note the class of *'open area'* dungeons, and list a number of games with representations of three-dimensional dungeons: either because they are multi-level 2D layouts, or realised in a fully 3D world. Many modern 3D game levels include a 'spatial puzzle quality' and as such share many design considerations with basic or even complex dungeons. Similar design considerations are also present in certain open-world games, set against a context of greater uncertainty with regards to the player's skills and abilities. Development work towards dungeon generators may be helpful for future generation of content for 3D levels more generally and even spaces of open-world games.

Some of the specific expectations relating to dungeons (as distinct from levels or open-world areas more broadly) are conducive for procedural generation. Dungeons are finite, and self-contained, it can be assumed that the player's skills and abilities are known at the point when the enter the dungeon and will not change. These properties help to reduce the number of dimensions and their cardinality in the design space relating to dungeons, and set a baseline for tractable generation.

The common structural elements of the dungeon concept, in combination with the favourable self-contained and finite properties, make them a natural target for procedural generation. Many of the more complex or cosmetic qualities may be set aside to facilitate an initial approach to generation. In the next section, we investigate the key element that defines the structural 'puzzle' dungeons present, followed by an overview of existing approaches to dungeon generation and a dissection of a useful generation pattern.

### 5.1.1 Key and lock puzzles

Dungeons in games have a structure that is influenced by spatially-separated locks and keys, which can only be passed in a specific temporal partial-order. This structure is part of what makes them an interesting generation challenge. Dormans (2017) presents an initial taxonomy of key and lock properties. Both keys and locks may have a range of different properties that affect how they can be used within the dungeon, and therefore what effect they should have

| | Strictness | Permanence | Directionality | Safety |
|---|---|---|---|---|
| Locks | conditional<br>dangerous<br>uncertain | permanent<br>reversible<br>temporary<br>collapsing | bidirectional<br>asymmetrical<br>valve | safe<br>unsafe |

Table 5.1: Properties of locks according to the taxonomy by Dormans (2017).

on the dungeon's structure. Keys and locks may have some additional properties that have no effect on the dungeon's structure — such as a connection to the wider narrative, cosmetic variants or local mini-games/skill-based usage. In addition to key-and-lock challenges, there are local challenges that have no effect on the dungeon's structure. The presence of non-strucutral properties and challenges may be useful ancillary information within the generation process (for the purposes of variety or continuity) but do not represent hard constraints. There are other advanced but non-essential concerns relating to structure (valves, shortcuts, the traditional return-home) that are beyond the scope of the current work but represent interesting future research directions (Sec. 5.4, Chapter 7). Of interest to the current work are hard constraints about structure that can make a dungeon impossible to complete if violated, plus selected soft constraints relating to good dungeon design. In conclusion, the structural constraints imposed by the key-and-lock model of dungeon design make the domain well suited to generation by ASP, and the existence in the domain of generation approaches using other techniques may be useful for evaluation purposes (Chapter 6).

Dungeons have a start, a series of challenges and an end, much like many game levels more generally. However, unlike game levels which may be linear (presenting only one path for progression), dungeons present choices about alternate routes forward. Often, progress along one route may turn out to be blocked by an obstacle of some kind (hereafter called a 'lock') until the player has found an appropriate object on some other route (hereafter called a 'key') that grants them the ability to bypass the initial obstacle. Aside from for tutorial purposes, the physical separation of the areas containing the lock and the key is typically part of the experience — the player is forced to explore and often to overcome intervening minor challenges in order to unlock the ability to progress. In a well formed dungeon, the player is able to explore and find a suitable key for each lock they encounter, potentially multiple times until they are able to construct a clear route to the end of the dungeon. Though commonly represented in games as literal keys and locked doors, this same concept may cosmetically be represented in multiple different ways: the 'key' may simply be a lever that opens a door in a different area, or the 'lock' may be a gate that is rusted shut and requires oil and a crowbar found elsewhere in the dungeon to open it.

Dormans (2017, pp.91–93) presents a taxonomy of possible key and lock properties that determine the constraints they impose on playability, of which two are presently relevant: keys may be either *consumable* or *persistent* (i.e. able to be used once only or multiple times) and may be *particular* or *non-particular*: able to open a single, specific lock, or any of a class of available locks respectively.

| | Persistence | Particularity | Usage | Portability | Safety |
|---|---|---|---|---|---|
| Keys | persistent<br>limited<br>consumable | particular<br>nonparticular | single-purpose<br>multipurpose | portable<br>assumed<br>fixed | safe<br>unsafe |

Table 5.2: Expanded properties of keys based on the taxonomy by Dormans (2017).

Many 'key' items in games are not represented by literal keys but some collectable object that solves or bypasses an obstacle elsewhere in the play area. The specific nature of the key items and their associated obstacles are generally closely tied to individual games, but as their overall behaviour is the same we can represent them during generation simply as *non-consumable, particular* keys. Another element of Dorman's taxonomy refers to whether keys are *single-purpose* or *multipurpose*, this is a related but nonidentical concept.

Dungeons in the 'Zelda' style contain two kinds of challenges: key-and-lock combinations that relate to the order of progression through the dungeon and require visiting two or more specific locations in a certain sequence, as described above, and also localised challenges that take place entirely within a single small region ('room') of the dungeon. We define such 'local' challenges as those that may be completed using only abilities that the player character may be assumed to possess at the start of any level plus those that are available within the room itself, and they are to a degree interchangeable without affecting the dungeon structure. For this prototype we consider three kinds of local challenge: *combat*, *puzzle* and *traversal*. We presently distinguish these only to ensure a degree of balance and variety among these different kinds; however annotation of room nodes with these concepts could be used for future refinement by a human designer or procedural generator.

In contrast, 'non-local' concepts impose constraints on the structure of and critical path through the generated dungeon. They connect physically distant parts of the structure graph via a partial temporal ordering that informs generation decisions (Listings 2-8).

Keys and locks form an important part of the hard constraints relating to dungeon playability: in a badly designed or generated dungeon a player might be unable to progress far enough to reach the boss if they cannot find or use the correct keys (see Fig. 5-4; Sec. 5.1.4).

### 5.1.2 Dungeon generation

Since the early days of computer gaming, dungeons and similar spaces within action-adventure games like *Rogue*, *Hack* and their descendants have been procedurally generated, often in highly bespoke and game-specific ways (Valtchanov and Brown, 2012; Dahlskog, Björk and Togelius, 2015; Shaker, Togelius and Nelson, 2016; Dormans, 2017). Several previous approaches to dungeon generation for 2D action-adventure games in academic literature have used graph-rewrite rules and spatial grammars to develop an initial model of the 'mission' within the dungeon (sequence of user actions required for completion) and then further rewrite rules to develop a gameplay space that supports the execution of that mission (Dormans, 2011; van der Linden,

Lopes and Bidarra, 2013; Karavolos, Bouwer and Bidarra, 2015; Lavender, 2016; Dormans, 2017). However, the presence of *hard playability constraints* (e.g lock and key puzzles) within this domain suggest that a constraint satisfaction approach for procedural generation as described by Smith and Mateas (2011) may also be effective. Previous work in this area has successfully demonstrated the use of Answer Set Programming (ASP) for generation of simple dungeons or similar spaces directly within a small tiled grid Nelson and Smith (2016); Smith and Bryson (2014); Neufeld, Mostaghim and Perez-Liebana (2015). We propose a combination of the graph and constraint approaches that instead produces dungeon level models in an abstracted graph form via constraint satisfaction, by modelling the graph generation problem and associated constraints as an ASP problem formulation and using a domain-independent solver to extract valid models, as in Smith et al. (2012).

The simplification afforded by abstracting a dungeon layout as a graph enables a number of possible approaches to generating dungeons using a graph as a starting point. A previous survey on the topic by van der Linden, Lopes and Bidarra (2014) lists a range of techniques and their implementations in literature; many of those most relevant to the present work are described in Sec. 2.2.1.

Overall we sketch a new approach for producing mission graphs for dungeon levels that differs from the graph-rewriting techniques in literature, and instead builds on Smith and Mateas's work Smith and Mateas (2011), using ASP to generate models of content that fulfil the range of important gameplay and design criteria present in this domain. We demonstrate an application of answer set solving as a generative method for dungeon level models in 'Zelda'-like action-adventure games, and provide a comparison with existing work using an expressive range analysis (in Chapter 6).

### 5.1.3 Top-down generation approach

Content and structure in games exist at a range of scales (Hendrikx et al., 2013). Human designers operate across multiple scales simultaneously, and aim to understand that changing certain elements at one scale will have knock-on effects at another scale. In contrast, generators typically operate at a single scale at a time, and attempt to build either upwards or downwards. Bottom-up generators operate by assembling components of the smallest scale or the lowest level, and then attempting to build a suitable high-level structure from these components — Smith and Bryson (2014) operates in this manner. In contrast, top-down generators begin by producing an abstract high-level representation of some aspect of the desired content, and then repeatedly refining that model in ways that add additional detail until it is sufficiently complete to serve as a usable description of the generated content; this is the core concept of Dormans' entire oeuvre of mission-space duality research.

Though not a technology per se, one common approach to Procedural Content Generation (PCG) is the process of abstract model refinement. At a basic level, this is a multi-stage generation approach which, rather than generating complete content in a single pass, constructs successively more detailed models of the content over multiple iterations — each of which may

make use of differing forms of PCG in order to improve the model. This is analogous to the typical human designer approach of defining in an informal, high-level manner which elements of an area are connected, and then 'greyboxing' layouts with low-fidelity representations of the final content (often, literally grey boxes) in order to assess which areas feel right and may be developed further, and which areas need to be reworked.

A number of projects previously covered elsewhere in this thesis use elements of this approach. Togelius, Justinussen and Hartzen (2012) use genetic algorithms to produce abstract descriptions of basic dungeon levels, which are then 'reified' (converted to usable content) by an ASP model taking the description as input. Both Smelik et al. (2010) and Liapis, Yannakakis and Togelius (2013b) use the concept of 'map sketches', which may be generated or produced by hand, to provide coarse abstractions of game environments. In comparison to final content, they are a higher-level description designed to be easy to produce and evaluate, and small enough to optimise and improve using automated techniques. A range of noise and simple simulation algorithms are used to convert them into final output. Finally, Smith and Bryson (2014) use the approach to constrain the domain of their ASP dungeon generation implementation. The first pass defines a spatial layout of rooms and corridors based on supplied content 'chunks', and a possible second pass populates these with treasure, enemies and other features.

Some projects use model refinement in order to generate one class of content based upon another. An obvious example of this is the provision of a generated environment to provide locations for a generated mission structure, as described by Dormans (2011). They note the essential duality between 'mission' and 'space' in many games and therefore the necessary dependencies that must be considered when generating both together, and present a system that treats the generated mission structure as a model of the level to be generated. An implementation of this system is provided by Lavender and Thompson (2015), who also assess its performance using the methods described in Chapter 6.

The iterative nature of the approach enables natural integration of opportunities for mixed-initiative production of content. Karavolos, Bouwer and Bidarra (2015) describe a multi-step grammar-based approach for generating 2D platform-game levels or 2D action/adventure dungeon levels. The first stage of the system uses a graph grammar to define connections between nodes such as 'Obstacle', 'Enemy' or 'Reward'; later stages use domain-specific spatial grammars to convert these nodes and connections into either platforms and pitfalls or rooms and corridors. Finally, concrete in-game representations of these elements are selected from a library of pre-produced 'chunks' of content that fulfil the properties specified by terminal symbols in the shape grammar. At any stage, a designer may alter the current model of the content, editing connections, node types or representation selection, to alter the system's final output. These edits are treated by the system as fixed, and so it attempts to further reify the model in a manner that is consistent with them[1].

[1] in these contexts the term 'reify' is used to the process of converting an abstract specification into an 'equivalent' in some sense concrete playable artefact. ASP outputs a set of internally-consistent facts; an abstract description of the content, about which specific choices must be made in order to end up with a concrete representation in a game. There are multiple ways that one could reify a connectivity/structure graph for example, but all of those outputs could still be described by the same abstraction.

As can be seen from the literature presented so far, research into procedural content generation is an active area. According to a survey on *Procedural Generation of Dungeons*:

> "In general, what current procedural dungeon generation methods are missing is not performance, but more powerful, accurate and richer control over the generation process."
>
> van der Linden, Lopes and Bidarra (2014)

### 5.1.4 Boss Key transcription approach

To illustrate the non-local relationships in dungeons within this chapter we make use of and extend Mark Brown's 'Boss Key' representation for dungeon lock and key arrangements (Figs. 5-3, 5-4 and 5-8), which was developed to communicate design observations about Zelda dungeon layouts and high level spatiotemporal structures as part of an educational video series (Brown, 2017). Though it bears similarities to the earlier graph representations used by Dormans (2011, 2010) and others covered in Sec. 2.2.1, it is not constructed as an intermediate step for generation but rather as an explanatory tool to address the problems that arise when analysing traditional 2D maps of existing dungeons. The approach simplifies consideration of progression through a dungeon by discarding all local challenges and most spatial information, and considering only the structure formed by basic connectivity between elements of non-local challenges in the dungeon.

It can be unclear from a purely spatial overview of a dungeon which areas are accessible based on changes in player abilities and possessions during the course of the dungeon. The format is therefore designed to omit unnecessary information; clearly map temporal progress in addition to spatial relations, and thereby highlight otherwise difficult-to-read information such as the degree of choice available at different points or the necessity of backtracking through previous areas. Icons are used to represent instances of specific feature classes such as locks or keys (Fig. 5-3). Vertical layout is used to indicate a partial order on temporal progression, which helps to indicate when specific portions of the dungeon are accessible. The horizontal axis represents parallel exploration possibilities, with relative locations along the vertical axis indicating necessary preconditions. Though this is intended as a descriptive approach for visualisation purposes, we propose that it may also be a useful level of abstraction to reason at for the purposes of dungeon generation, and for identifying malformed outputs (Fig. 5-4). Generating graphs of this form is a first step towards generating dungeons, and they can later be refined by introducing local challenges, which are by their nature guaranteed not to break the structure. The graph does not necessarily have a direct relation to the final dungeon, and is conceptually similar to Dormans' mission-space duality (Fig. 2-1). Overall, the Boss Key diagrams are a useful abstraction of the spatiotemporal structure of dungeons, show promising potential as an intermediate abstraction for performing top-down generation of dungeon structure graphs.

Fig. 5-3 shows the simplest possible map containing all of the non-local concepts listed previously (actual dungeons can often be significantly more complex; cf. Brown (2017)). The

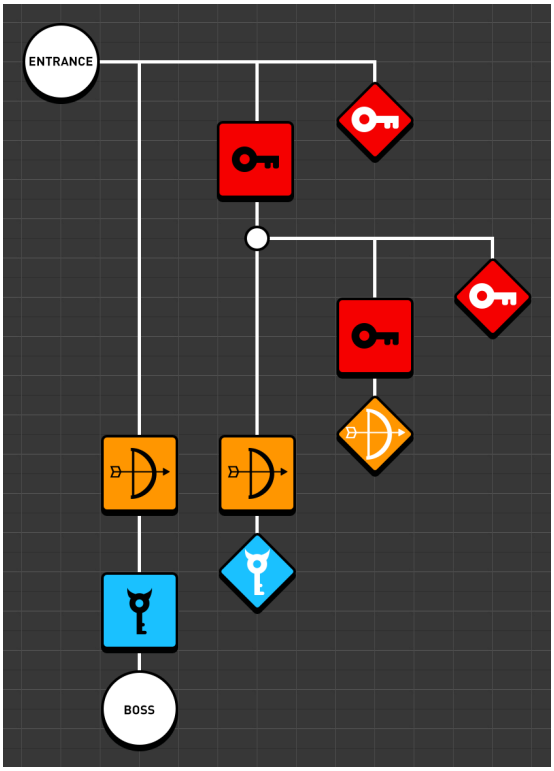



Figure 5-2: The initial example, a Boss Key diagram of Gnarled Root Dungeon (Fig. 5-1). Reproduced from Brown (2017).

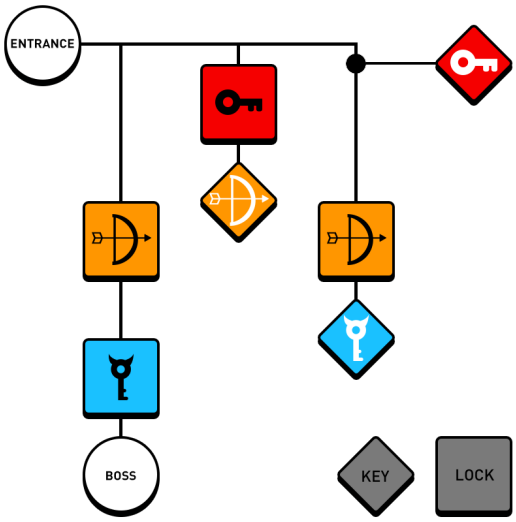



Figure 5-3: Sample layout of key (diamond) and lock (square) concepts in Boss Key style (Sec. 5.1.4; image assets from Brown (2017)). Concepts are related by colour and icon, connections are traversable paths (abstracting away non-local challenges), and relative vertical positions give a temporal partial order.

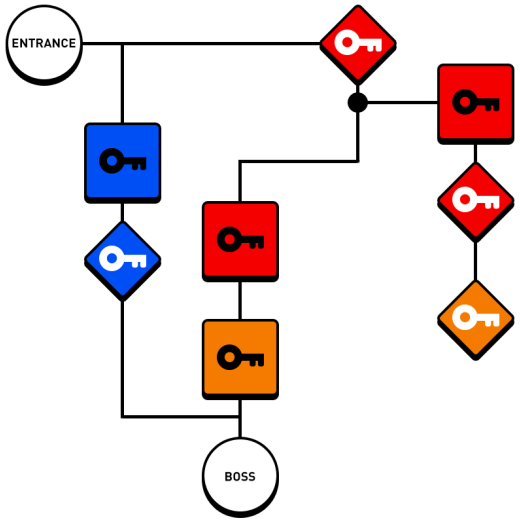



Figure 5-4: A pair of undesirable challenge arrangements mapped using the Boss Key approach. Left branch: a key is inaccessible behind its associated lock. Right branch: the first red key could be used in the wrong (left) lock, leaving other keys inaccessible (see Sec. 5.1.4).

lines connecting nodes represent concepts accessible to the player, and their temporal progress through the dungeon is roughly mapped down the vertical direction of the diagram. From the Entrance of the dungeon, one Small Key and three locks are initially accessible (though the two Dungeon Item locks cannot be opened until later). Once acquired, that Small Key can be consumed to open the matching (central) lock, granting access to the Dungeon Item (in this case, a bow and arrow). The player may then open both Dungeon Item Locks in either order (perhaps by firing arrows at inaccessible levers controlling gateways), providing passage to the Boss Key and the Boss Lock. These may be accessed each in turn to reach the Boss itself, and the end of the dungeon.

In contrast, Fig. 5-4 illustrates some possible violations of typical dungeon layout constraints. The most obvious is that the arrangement of the blue lock and key on the leftmost branch is reversed, meaning the player is unable to open the lock using the key they cannot access behind it. A more subtle issue is present on the right branch, where it is possible for the player to make choices that leave the dungeon in an incompletable state. If the player chooses to use the red Small Key to open the left red lock rather than the right one, then they are once again in a situation where the key they need next is inaccessible behind the lock it would open on the rightmost branch. Though some games are designed with mechanics that reduce the impact of otherwise incompletable levels (Dormans, 2017), in general it is desirable to ensure that these situations and others like them cannot arise.

This style of dungeon diagram is a high-level representation of the overall progression of accessible subsections of the dungeon. Notably, it does not include any information about local obstacles within the dungeon, nor does it show any route, key or lock that is not on the critical path between the Entrance and Boss nodes. However, it is useful for illustrating the

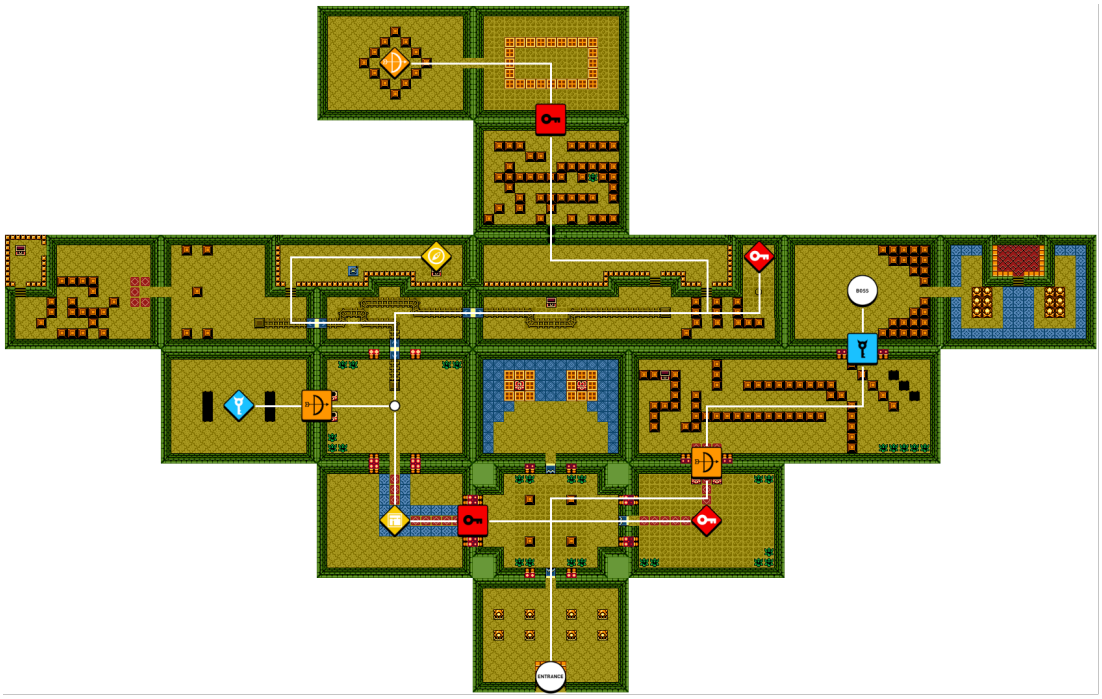


Figure 5-5: The dungeon presented in Fig. 5-1 overlaid with the Boss Key graph from Fig. 5-2, showing the indirect relationship between mission and space as in Fig. 2-1.

relationships between non-local challenges, comparing the spatiotemporal structure of two or more visually-dissimilar dungeons, and many of the original dungeons from the Zelda franchise have already been mapped in this style and are available online Brown (2017).

## 5.2 Dungeon graph generation

In this section we describe the ASP-driven approach used to generate dungeon graphs, as an initial step towards providing a design assistance tool for level greyboxing. To start we define terms relating to the problem space — given the variety of dungeons, generators and features that have been discussed in the preceding sections, we draw out some of the common core concepts that constitute the essence of the intended generation domain. Then we briefly describe an initial attempt at ASP formulation of the problem and the issues it highlighted, followed by a detailed investigation of a more complete formulation that satisfies our intentions. Finally, we present outputs from the system and discuss their merits and the ways in which the system could be altered.

### 5.2.1 Problem outline

In this section, we specify our aims for the dungeon generator system. We restrict our scope to the generation of Zelda style dungeons as they contain a number of commonly-understood design

tropes. In addition, the presence of existing generators in this genre facilitate comparative evaluation (as in Chapter 6). We provide an overview of the terminology for the classes of key-and-lock puzzles we intend to consider, and other important features, and explain and discount certain aspects of dungeons in some games to restrict initial scope. In the following sections we discuss how we use this definition to inform development and evolution of an ASP encoding of the problem.

In this paper we will be specifically considering constraints patterned on those found in the 'Zelda' franchise of games, which contain three notable classes of keys per dungeon:

- one or more *non-particular consumable* keys (known as Small Keys) and an equal number of associated locks, potentially resulting in a choice of how to progress (see discussion below);
- a Boss Lock immediately before the Boss, only unlocked by a *particular consumable* Boss Key elsewhere in the dungeon; and
- a Dungeon Item, a *persistent*, *non-particular* key-like item that allows the player to cross or 'unlock' multiple previously impassable lock-like obstacles blocking the route to the Boss Lock / Boss Key.

The features described in this section each indicate constraints that must be satisfied if we want to generate dungeons in this style, however for this prototype we use modified versions of two of the 'Zelda' restrictions: 1) as in the shape-generation portion of Lavender (2016) we consider only acyclic dungeon layouts, to assist direct comparison and simplify implementation, and similarly 2) we start with an implementation of the Small Key concept that is *particular*, and therefore potentially provides fewer choices. Many existing Zelda levels are within the space represented by these modified constraints, and so we consider them acceptable for illustrative purposes.

### 5.2.2 Initial formulation

```
event(0,start). event(1,puzzle). event(2,combat). event(3,traversal).
event(4,traversal). event(5,combat). event(6,combat). event(7,get_key).
event(8,puzzle). event(9,use_key). event(10,end).
```

Listing 5.1: Simple schedule facts output from parameterised outline generator.

To guide later stages in the generation, a linear abstract model is generated according to constraints on node types and feasibility of basic lock and key interactions. Each possible distinct event within a single playthrough is ordered.

We start by generating an abstract model of the content to be generated: in this case, a 'schedule' for events that a player following the shortest path through the level must complete. We define within the first ASP program a pool of possible events such as combat or traversal challenge, that the player can experience during the course of a single level. From this pool we

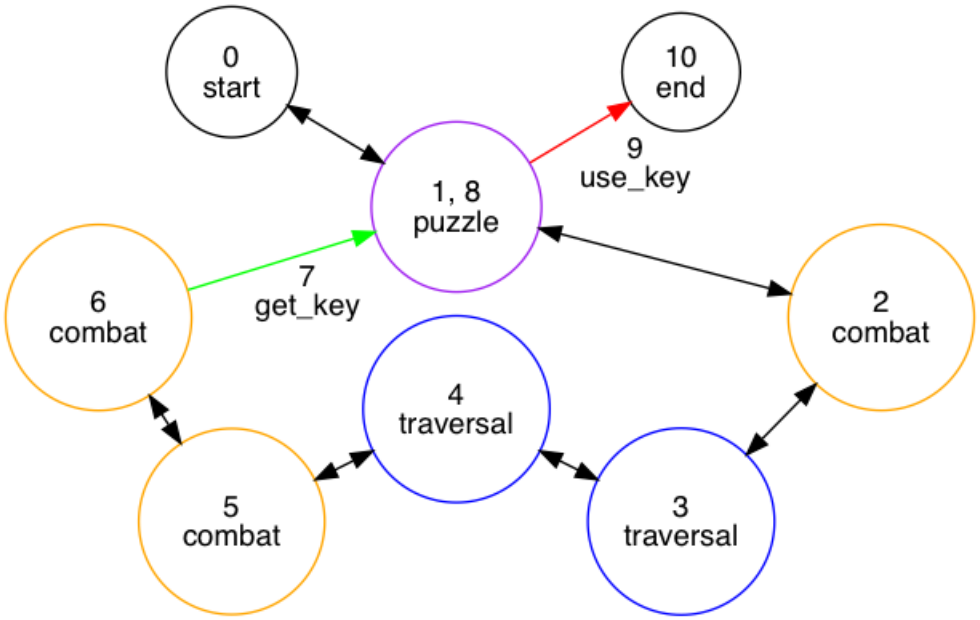



Figure 5-6: Directed graph of level concepts from schedule, with local concepts on nodes and structural edges.

provide rules to generate a sequence of appropriate length, constrained according to expressed requirements — for example, forbidding the selection of the same event more than twice in a row, or selecting a `use_key` event before any `get_key` events have occurred. Clingo 5.2.0 is used to ground and solve the problem[2], and a single answer set is randomly selected as the abstract model to be further refined in later steps. The answer sets of this initial program are presently emitted in a form similar to that in Listing 4[3] though there is scope for further information such as challenge subtypes or narrative events to be generated at this stage.

Once an appropriate level outline has been selected, a new ASP program uses facts regarding the outline as input to then solve for a directed graph representation of the critical path within the level. The trivial conversion would result in a new area generated for each successive event, however at this stage the formulation also allows re-use of previously-visited areas for challenges of the same area type, and supports the introduction of unidirectional 'valve' shortcuts to return to these re-used areas. The use of ASP makes it easy to declaratively integrate constraints relating to connectivity within this generation step, ensuring that if the problem formulation is correctly constructed, the output will be also.

Figure 5-6 shows an outline of a simple level model using these concepts. The schedule in Listing 4 is laid out as a loop where the first (puzzle) challenge area offers an impassable connection to the level end, but the player must first complete a series of other challenges before acquiring a key and returning to this initial area, to complete a new puzzle, use the key

[2]configured with `"-seed=$RANDOM -sign-def=rnd"` to provide additional variance between successively-emitted answer sets.

[3]with some post-processing: answer sets are emitted as facts without terminal '.'; addition whenever bracket count is 0 results in valid ASP syntax that we can insert back into following stages.

and complete the level. This particular layout of the schedule demonstrates the design pattern where players are allowed to encounter the problem (the key required at `9`) before the solution due to the re-use of area `1`, but alternate layouts could instead or also have reused area `2` for the combat in `5` or `6` (not both: areas may not be immediately re-used).

The level of detail contained in the initial schedule model, or pool of event types available, may be changed without significant alterations needed to the second step. Further specificity could allow for explicit identification of simple design patterns present in the schedule, such as 'three or more 'jump' traversal challenges, each with increasing length'. Design patterns could also be identified at the graph stage, such as 'hub' areas that are frequently revisited.

As the number of events and areas under consideration increases, it is possible that a generated directed graph would be non-planar and therefore potentially unsuitable as an output. The low average arity of these graphs means this is unlikely, however the feed-forward nature of the approach introduces a natural opportunity for external planarity testing of the selected graph before progression to the next step, with fallback to any other answer set in case of non-planarity.

Though the player's experience of a single playthrough of a level will necessarily be linear, in accordance with the initial abstract model generated above, the actual play space may potentially support multiple possible alternative routes. Multiple events within a single playthrough may occur in the same location at distinct times, meaning that the mapping from playthrough events to play locations must be surjective according to the compatibility of varying events with the available locations. In addition to the variance in event types, it may also be desirable to prefer semantically-guided variance in elements such as frequency of area-reuse and branching factor of possible paths; optional dead-ends and loops; optional rewards, etc.

The system as described represents the first two stages of a possible multi-stage iterative refinement approach. The third step would involve elaboration of the critical-path graph-map with optional areas, alternative paths and non-critical rewards. In the level as currently generated, after each event the player is presented with only two choices: continue to the next event in the sequence, or backtrack through previously-completed areas. At this stage in level development, a human designer would endeavour to add alternative choices to make the level feel less linear, without breaking the existing level by inadvertently adding shortcuts that allow the player to bypass critical content, or areas that allow the player to enter a dead-end state from which they are unable to progress. The third ASP-based transformation must therefore add new optional or alternate areas to support exploration or optional subgoals within the level, without introducing a path from start to end that is shorter than the initial abstract schedule.

Prior work on ASP for PCG suggests a meta-ASP approach for ensuring the absence of undesirable shortcuts in puzzles (Smith, Butler and Popović, 2013), however as the apparent connectivity of an action-adventure level may change as the player acquires keys (representing both items and new abilities), event-calculus-based modelling of possible play traces across the elaborated graph may be necessary. An ASP-based approach to this is presented in the system *Ludocore* (Smith, Nelson and Mateas, 2010).

Other considerations relating to the graph elaboration step include the use of more sophisti-

cated representations of keys and edges. Minimal constraints in the existing system ensure that each key may only be used once in the first lock encountered; however many action-adventure games include keys that may be used multiple times, or only in a specific lock, or locks that need multiple keys. Likewise, edge types that are single-use only, are unidirectional until unlocked, or are unidirectional on the critical path, each introduce additional potential constraints and opportunities in the elaboration step. It may also be desirable to specifically constrain or at least identify the presence/absence of loops, dead-ends, significant backtracking or particular levels of choice branching factor for individual levels or areas.

### 5.2.3 Advanced formulation

In this section we outline the key elements of our revised, improved implementation, expand on certain details of the encoding, and present and discuss a sample output from the described formulation (Fig. 5-7).

Our initial formalisation attempts to capture the high-level design concerns and commonalities of the Zelda-like dungeon domain, as described in Sec. 5.2.1 and including specifically the exceptions relating to acyclicity of the dungeon graph structure, and *particular*ity of the Small Keys. We follow the approach laid out in Smith and Mateas (2011) and *construct* a design space through the use of choice rules that generate a selection of available nodes within the graph, *deduce* additional elements of the design space through the use of deduction rules that infer additional necessary nodes, relationships and semantic tagging, and then constrain the design space through the use of integrity constraints that *forbid* undesirable outcomes.

Key elements that are produced by the initial choice rules are the initial pool of nodes and the parenthood relation assignment, which together form the basic structure of the graph. Some semantic tags provided by the deduction rules are inherent and listed in Table 5.3, such as the quality of being a `keyy(N)` or the `strt(N)`; others are relational and dependent on the assigned parenthood relation. Some integrity constraints represent concerns that are important for gameplay, while others are a matter of designerly intent — in the present formulation they are treated equally.

Within the choice rule for each directly-generated node type are specified an upper bound and lower bound on expected node counts as listed in Table 5.3; these initial values were selected observationally based on dungeon mappings in Brown (2017) for non-local concepts, and by inspection of existing dungeons for local challenges. The table also specifies the additional tags each node receives, the implications of which are detailed below.

To facilitate comparison with existing work in the domain using expressive range analysis, and to simplify both implementation and evaluation we presently consider only dungeons in the form of trees, which may require backtracking but do not contain connections between branches[4]. To efficiently guess a total, acyclic connection, each node is assigned precisely one parent[5] according to the `paft/2` or 'physically after' relation. To begin the process the node

[4]Adding alternate routes, cyclic routes as in Dormans (2017) and/or shortcuts is left for future work.
[5]mostly. The `strt` node is a special case.

| name | ID | min | max | local | tags |
|---|---|---|---|---|---|
| Start | `start` | 1 | 1 | * | `strt` |
| Combat | `c` | 2 | 5 | * | - |
| Puzzle | `p` | 2 | 5 | * | - |
| Traversal | `t` | 2 | 5 | * | - |
| Small Key | `sk` | - | - | | `keyy, rewd` |
| Small Lock | `sl` | 1 | 3 | | `lock` |
| Dungeon Item | `di` | 1 | 1 | | `keyy, rewd` |
| Dungeon Lock | `dl` | 2 | 3 | | `lock` |
| Boss Key | `bk` | 1 | 1 | | `keyy, rewd` |
| Boss Lock | `bl` | 1 | 1 | | `lock` |
| Reward | `r` | 2 | 5 | * | `rewd, <special>` |
| Boss | `boss` | 1 | 1 | * | `rewd, critical` |

Table 5.3: Initial configuration of bounds and semantic tags.

labelled `strt` is assigned as its own parent, with a fact stating that `strt` is physically after itself:

```
paft(N,N) :- strt(N).
1{ paft(B, N) :paft(_, B) }1 :- node(N), not strt(N).
```
L.5.2

Thereafter any node `N` that is not labelled as the `strt` is assigned precisely one parent `B` from among atoms that are already a child in a `paft` relationship. This ensures that we generate a single valid tree containing a route through the dungeon and all nodes are ultimately connected to the `strt`. Integrity constraints can be used to disallow undesirable outcomes; for example forbid any answer set where some node has the `boss` or the boss key as its parent, or the `boss` is not behind a lock:

```
%% boss and bosskey must both be terminal
:- node(N), paft(boss, N).
:- node(N), paft(bk(1), N).

%% boss must be behind bosslock
:- node(boss), not paft(bl(1), boss).
```
L.5.3

A similar approach can be used to tag all nodes that represent dead-ends within the graph (have no known children), and then forbid all answer sets where those nodes are not tagged as a reward — this ensures that the generated dungeon will never contain useless dead ends where a challenge leads to no payoff:

```
terminal(N) :- node(N), not paft(N, _).
:- terminal(N), not rewd(N).
```
L.5.4

For non-local concepts there is also a `taft/2` or 'temporally after' relation; represented

explicitly in Fig. 5-7 by the dashed edges connecting keys to locks, and in Fig. 5-8 implicitly via relationships between vertical heights, and colour-/symbol-coordination. The union of `taft` and `paft` represents a partial order across the nodes in the graph, with `start` and `boss` at first and last respectively.

```
%% trace criticality
critical(boss).
critical(N) :- paft(N, P), critical(P).
critical(N) :- taft(N, T), critical(T).

%% restrict deviation
exploration(N) :- node(N), not critical(N).
:- 5{exploration(N) :node(N) }.
```

L.5.5

Deduction rules allow us to selectively apply additional semantic tags to nodes, and identify routes that are not on the critical path to the `boss`. These are greyed out in Fig. 5-7 and faded in Fig. 5-8 to signify that they are optional — though the constraint in Listing 4 ensures that any optional path will necessarily be rewarding.

```
%% locks imply the existence of their key
node(sk(X)) :- node(sl(X)).
taft(sk(X), sl(X)) :- node(sk(X)), node(sl(X)).
keyy(sk(X)) :- node(sk(X)).
lock(sl(X)) :- node(sl(X)).

%% lock cannot be immediately after another lock
:- paft(X, Y), lock(X), lock(Y).
```

L.5.6

Small Locks, Dungeon Item Locks and the Boss Lock are all part of a `lock/1` category with certain commonalities: e.g. there are never two in a row without some other concept in between; locks are never physically before their own key. Likewise, Small Keys, the Dungeon Item and the Boss Key are all part of the `keyy/1` category.

This formulation of the dungeon generation problem within ASP occupies 50 lines of code, not counting whitespace or comments. We use Clingo 5.2.1[6] via Python, and configure the solver with `solver.sign_def = "rnd"` and a random seed from numpy.

### 5.2.4 Sample generated dungeon graphs

Output from the ASP is a series of facts relating to a model of the dungeon, which the Python script translates into a source format suitable for rendering with GraphViz; one example is shown in Fig. 5-7, with a Boss Key equivalent in Fig. 5-8. Several instances of the concepts represented by Listings 1-8 are apparent. Clingo returns a new model in considerably less than

[6] `https://potassco.org/clingo` — accessed 27 November 2020

one second[7], which facilitates casual experimentation with alternative formulations or varying parameters.

Rendering a single sample output in this way can be instructive while attempting to refine the formulation as it allows easy observation of potentially undesirable outcomes, however without more thorough analysis (such as is covered in Chapter 6) it can be difficult to know whether any single flawed production is representative of the generator's typical output. As an example, in the sample output, the leftmost branch contains two key-like items (the Dungeon Item and Boss Key) with no challenges in between. If this is deemed undesirable, there are two possible solutions suggested in Listing 9.

```
:- paft(A, B), keyy(A), keyy(B).                                    (9)
:- paft(A, bk), not local(A).
```

The first rule is modelled on the equivalent formulation for `lock`s that forbids any two in a row (Listing 8). The second approach is particular to only the `BossKey` and ensures that it is always preceded by a local concept. Either alteration could fix this specific arrangement, though it is also undesirable to specify too many special-case rules. (One of the key advantages of the declarative constraint-based approach is the ease with which it can be remodelled; graph grammars are described as 'unwieldy' and 'fragile'. In general, it is probably preferable to choose minimal, systemic solutions to problems.) However, this rapid iteration and refinement of the space of possible outputs is a demonstration of the iterative process proposed by Smith and Mateas (2011).

The system as presented is an improvement on the formulation discussed in Sec. 5.2.2, that explicitly reasons over connectivity, temporal relations for non-local concepts, semantic tagging for ease of declarative constraints (e.g. the critical path/exploration dichotomy and forbidding successive duplicates), and appropriate population thresholds informed by observation.

## 5.3 Unreal Engine integration

As many studios (including the research partner placement company, Ninja Theory) make use of *Unreal Engine 4* (UE4; Epic Games, 2004), a version of the Clingo library was built from source as part of a custom plugin for UE4 and made accessible from the visual scripting language, Blueprint. A simplified API is presented that can be instructed to load ASP files, solve, and emit output in JSON to be parsed and processed by the individual game in a domain-appropriate manner.

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. Our implementation integrates the monolithic ASP system *Clingo*[8] (Gebser et al., 2014) with the commercial games development platform *Unreal Engine 4*[9] (UE4). Clingo is built as a shared library plug-in for

[7] or 10,000 in less than 6s on a 6-core 3.7GHz Windows 10 PC with 16GB RAM.
[8] `https://github.com/potassco` — Clingo and related tools, accessed 27 November 2020
[9] `https://www.unrealengine.com/en-US/` — Unreal Engine 4, accessed 27 November 2020

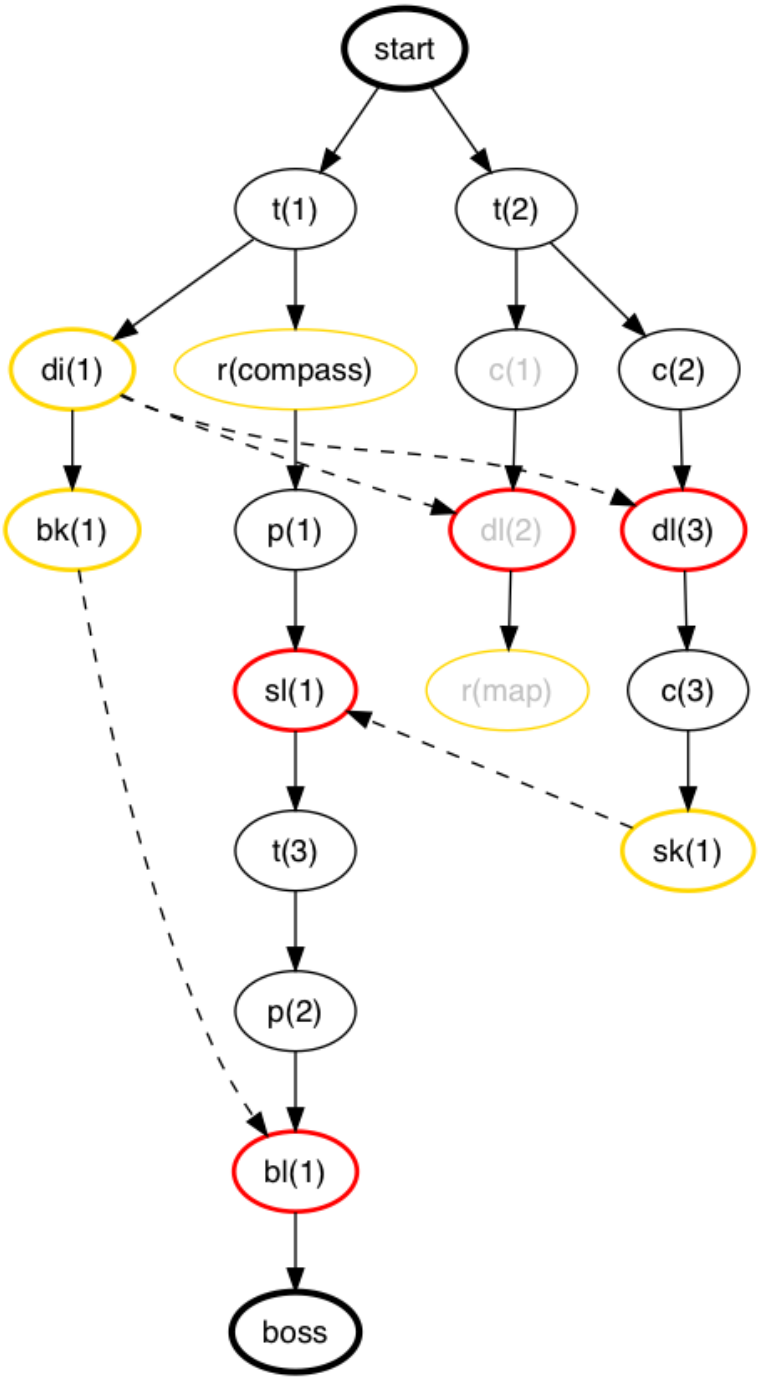



Figure 5-7: Example of a generated graph with local (black outline) and non-local (bold, coloured red/gold) challenges, rewards (gold), optional path (grey text starting at `c(1)`) and explicit temporal relations (dashed connections). A Boss Key abstraction of this graph is presented in Fig. 5-8 for comparison, and node IDs are detailed in Table 5.3.

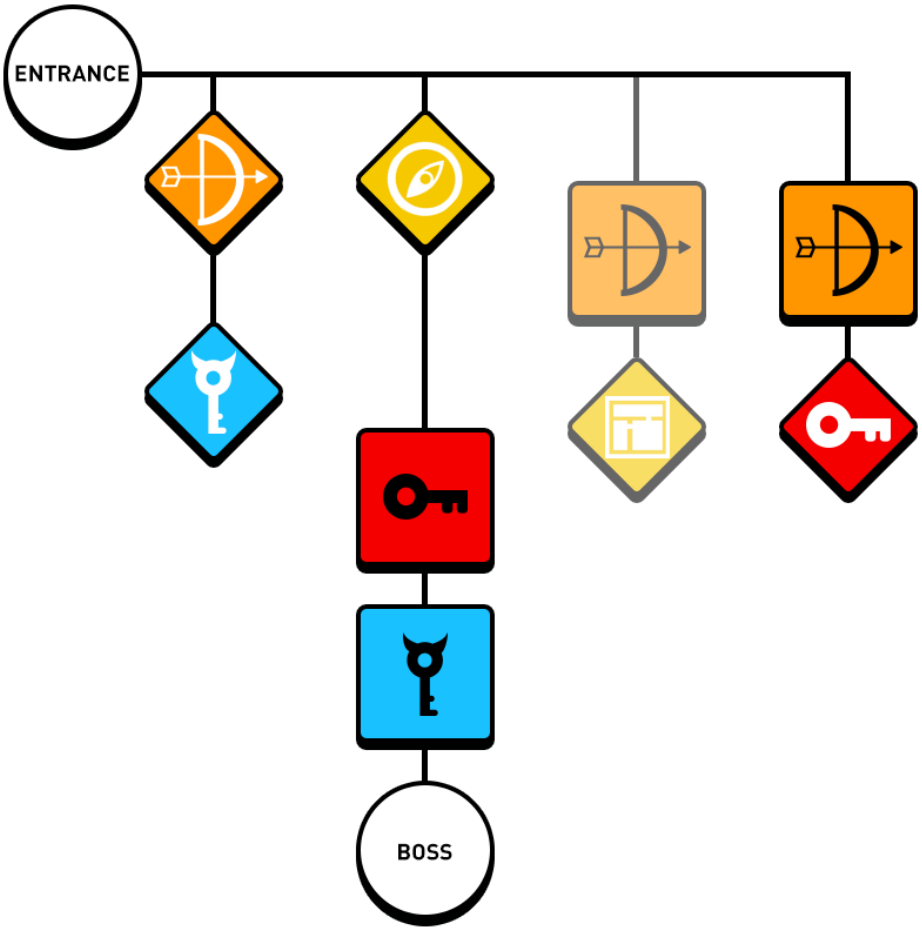



Figure 5-8: Boss Key abstraction of generated graph in Fig. 5-7, with only non-local challenges, optional path and rewards detailed.

Figure 5-9: The user interface of the UE4 editor (Epic Games, 2014).

UE4 and made accessible from the visual scripting language in order to load ASP files or code fragments, solve, and emit output in JavaScript Object Notation (JSON) to be parsed and processed by the individual game in a domain-appropriate manner.

Finally, to use this approach to generate usable level greyboxes, it is necessary to support solver output integration with a game engine and/or editor, and reify the abstract graph model into a concrete representation. Given a planar graph with low average arity, a number of assumptions can be made to simplify the embedding problem. Relevant hints for appropriate representations can be directly generated or inferred from the contents of the abstract model — for example a unidirectional 'valve' connection between areas could be embedded as a safe drop that that player cannot return up past. As an initial greybox, rooms containing local concepts are blocked out, and icons represent the presence and relationships of non-local structural concepts.

## 5.4 Discussion and further possibilities

The production of greybox sketches is a key stage in the design of action-adventure levels, which share many key concepts with levels in 2D action-adventure games generally. The generation of levels of this kind is a constrained, structural problem due to the presence of key and lock puzzles and potentially other designer-guided requirements. The BossKey representation allows for abstraction of the play graph to only non-local elements, simplifying generation, and it can be later re-populated with appropriate interstitial local concerns. Representation of these dungeon structure graphs is possible in ASP, which allows for quick generation and 'sculpting' of the

design space, under certain appropriate assumptions. We have integrated an implementation of this apprach with UE4 to allow quick iteration over possibilities

> "A Casual Creator is an interactive system that encourages the fast, confident, and pleasurable exploration of a possibility space, resulting in the creation or discovery of surprising new artifacts that bring feelings of pride, ownership, and creativity to the users that make them."
>
> Compton and Mateas (2015)

The 'casual creator' system definition above is relevant but not sufficient: we are building for expert users, but speed, confidence and pleasurable exploration are still useful goals. The present outcomes suggest a range of promising future work.

### 5.4.1 Representation sophistication

An initial improvement would be to increase the sophistication of constraints expressible on the system, such as reasoning over a wider range of concepts present in some traditional Zelda levels including shortcuts between branches or *non-particular* Small Keys. Additional desirable properties might include explicit reasoning over backtracking, the effects of reward items, or layout symmetry.

Other considerations relating to the graph elaboration step include the use of more sophisticated representations of keys and edges. Minimal constraints in the existing system ensure that each key may only be used once in the first lock encountered; however many action-adventure games include keys that may be used multiple times, or only in a specific lock, or locks that need multiple keys. Likewise, edge types that are single-use only, are unidirectional until unlocked, or are unidirectional on the critical path, each introduce additional potential constraints and opportunities in the elaboration step. It may also be desirable to specifically constrain or at least identify the presence/absence of loops, dead-ends, significant backtracking or particular levels of choice branching factor for individual levels or areas.

The current implementation is informed by assumptions common to many 2D action-adventure games: grid-based layout, regular rooms informed by screen size with four exits, etc. (levels exist in euclidean space, with no teleports or overlaps). However, the underlying graph representation is also amenable to representation on a less regular substrate, e.g. the Delauney triangulation of polygonal Voronoi-partitioned rooms; or flex grids similar Stålberg's work with Wave Function Collapse (WFC) on irregular grids (i.e. Townscaper).

### 5.4.2 Integration improvements

Several elements of the industrially-relevant engine integration could be iterated on in order to improve ease-of-use and user experience. At present, levels are directly greedily embedded into the 3D space once generated. In order to provide an automatable visualisation of the selected dungeon it would be beneficial to integrate a graph-drawing library and automatically generate the appropriate 'Boss Key' style diagram. When clingo is used via the python harness, dungeon

facts are automatically also translated into an appropriate Graphviz `.dot` format, however the current C++ plugin code only performs translation to JSON. One improvement would be to automatically render the structure graph of the current generated dungeon, or develop a custom renderer that can generate Boss Key diagrams from that information automatically. This could also serve as a step towards support for node-based mixed-initiative editing.

Further work currently includes developing a robust data-flow approach and fallback backtracking between ASP modules within UE4, including a designer-specified transformation scheme to guide progression between tasks. In order to support true mixed-initiative generation the system also needs to provide a method for converting in-editor alterations to content back into ASP facts representing those changes, to allow the solver to reason over them for iterative refinement and identification of constraint violations such as undesired short-cuts (Smith, Butler and Popović, 2013).

### 5.4.3 Complete game generation

Butler et al. (2013) propose a method for generating sequences of 'levels' (in their case, mathematical puzzles) containing an increasing number of concepts. A similar approach could be applied to the present work: if rather than generating levels in isolation a sequence of levels are generated according to a generated progression specification, additional functionalities are possible. Notably, Dungeon Item Locks relating to Dungeon Items that are known to have been collected in prior dungeons are then potentially available for use as local challenges (see Sec. 5.2.1).

Two further extensions to the iteration concept are also desirable, to generate orders of content that are both more abstract, and more concrete. Abstractly, prior generation of an overarching 'progression schedule' over multiple levels would allow both progression of concepts as described in Butler et al. (2013), but also specification that keys known to be found in earlier levels could fit locks in later ones, and specific challenges in later levels could be foreshadowed[10]. Concretely, information about an area relating to its connections, expected state of the player and required challenge(s) type could be used as input to a room refinement step similar to that in Smith and Bryson (2014), which dependent on context could involve ASP generation of suitable combat waves or a puzzle, or an integrated domain-specific room generation module.

### 5.4.4 Summary

> "Authoring, improving and maintaining grammars is difficult because it is hard to predict how each grammar rule impacts the overall level quality, and tool support is lacking."
>
> van Rozen and Heijn (2018)

[10] as in Unexplored: `https://store.steampowered.com/app/506870/Unexplored/` — accessed 27 November 2020

> "[...] the combinatorial explosion resulting from recursive rule expansions complicates forming mental models required for reasoning about intended qualities, and how they are represented in the grammar or intermediate data. [...] Grammars are brittle, i.e. code that is liable to break easily. Designers require special measures to ensure that qualities once introduced, remain intact, preventing successive rewrites from breaking levels."
>
> van Rozen and Heijn (2018)

In this chapter, we have described a system for procedurally generating action-adventure game level greyboxes using production via ASP constraint solving. This is a generation domain that has previously been addressed using other techniques, though the most commonly-attested in existing academic literature (grammar-based generation) is still only rarely used by industry. We present the refinement of a simple abstract model of a level (the events on shortest path to completion) into a more detailed abstract model of the level (a directed graph showing areas within the level by challenge type, and key and lock events), and discuss further steps including level elaboration for choice and optional areas, area refinement from known constraints, and integration with a game engine. We have also detailed a number of promising avenues for further development of this system. In the next chapter we address evaluation and iteration on the system as presented here.

# Chapter 6

# Dungeon Graph Evaluation

In this chapter we provide a quantitative analysis of the system described in Chapter 5. First, we introduce relevant terminology and an evaluation approach developed by Smith and Whitehead (2010) for visualising the characteristics of a generator's output, with examples of its prior application for other generation domains. Then, we discuss appropriate definitions for domain-general metrics that have been suggested by Lavender and Thompson (2015) for dungeon generation analysis, and the generative-grammar-based dungeon generator for which they were initially developed. Next, we present the results of the analysis performed on outputs from the system described in Sec. 5.2.3, and a direct comparison to the previous analysis undertaken by Lavender and Thompson (2015) on the outputs of their system, with discussion of the insights that this provides. Then we discuss how the outcome of this analysis can inform alterations to the system formulation, and demonstrate the effects of a small number of these changes. Finally, we conclude with a discussion of the utility of this evaluation approach, and some observations about both the analysis and the insights it provides about the system as presented.

A key problem with even automated visualisation of output as described in Sec. 6.1.2 and demonstrated in Fig. 5-7 is that it is difficult to ascertain how representative a single specimen is. This issue motivated Smith and Whitehead (2010) to develop an approach for characterising the 'expressive range' (variety and style, Smith (2017)) of a generator via visualisation of generator-independent quantitative metrics, sampled over a large number of outputs. A potential weakness of the approach is that this still isn't *complete* in any rigorous theoretical sense[1], and specific malformed instances may be missed by the sampling approach. However, as the sample size goes up so too does the confidence measure in coverage increase, assuming a sufficiently unbiased sampling strategy.

To assess the effectiveness of our approach and inform further development we perform an expressive range analysis as described by Smith and Whitehead (2010). This is a quantitative technique for visualising the variety and style of outputs a generator can produce (its 'expressive range'), by calculating and displaying the values of a small number of general metrics over a sample of outputs from the generator. We compare the initial visualisation with the analysis

[1] unless the generative space is sufficiently constrained that it is tractable to run a complete enumeration.

of another generator within the same domain (Lavender, 2016) and also with an updated visualisation resulting from minor modifications to the generator constraints, in order to illustrate both the resemblance in outcomes and capability for low-cost iteration through varied possible generator outputs.

## 6.1 Expressive Range Analysis

Proposed by Smith and Whitehead (2010), Expressive Range Analysis (ERA) considers the outputs of a generator in aggregate rather than individually. It is an approach that attempts to facilitate interrogation of the generator's range and responsiveness to changed parameters, and can also be used to compare an abstraction over the outputs of two or more generators within similar domains, as in Horn et al. (2014) where the technique is used to compare the expressive range of generators in the 2D platforming genre.

Lavender (2016) has already made use of the technique within the domain of Zelda-like dungeon generation to analyse her implementation using graph-rewrite rules, which provides a useful point of comparison with an alternative paradigm. We aim to generate comparable heatmaps across the same measures, in order to investigate the expressive range of the present system and compare its performance with another approach.

Intent: Measurable, global, emergent properties of the generated content.

Domain: Precise method is domain-specific, but metrics should be generator-independent.

### 6.1.1 Definition

The expressive range analysis consists of four main steps (Smith and Whitehead, 2010; Smith, 2017):

- Determine appropriate metrics. As we intend to contrast the outcome of this analysis with existing visualisations, we will use the same metrics, as defined below. In addition, we measure and report the average size of generated graphs.
- Generate content. We collect 1,000 individual sample dungeon models from separate seeds and collate the metrics scores for each.
- Visualise the generative space. We use matplotlib to render heatmaps of pairs of metrics (Fig. 6-2), comparable to the existing visualisation by Lavender (2016) (Fig. 6-3).
- Analyse the impact of parameters. In Secs 6.3, 6.3.1, 6.4 and Fig. 6-5 we compare and contrast the effects of slight alterations to the problem formulation.

Summerville (2018) provides an alternative definition for leniency: "the number of enemies plus the number of gaps minus the number of rewards" — for clarity in the present work we use the original definition from Smith and Whitehead (2010); the difference is mentioned here to illustrate the dangers of fragmentation/collision in terminology but mainly to be explicit and avoid confusion.

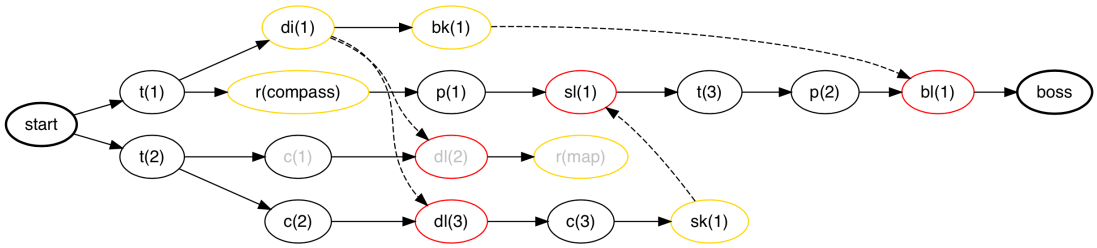



Figure 6-1: A structure diagram of one of the outputs from Sec. 5.2.4 reproduced here for comparison.

### 6.1.2 Application in other genres

This approach has been applied by Horn et al. (2014) to compare several generators and generator parameter configurations in the domain of 2d platformer level generation (Mario); as discussed in Sec. 2.5, see Fig. 2-4.

Summerville (2018) provide some additional techniques that are particularly appropriate for generators that 'learn' from original content, including a measure of plagiarism. They also motivate the use of corner plots for the presentation of ERA, describing them as "a visualization technique that allows for an arbitrary number of dimensions to be viewed holistically". Whitehead (2020) presents an application to a similar domain using a novel metric not necessarily applicable to the present work (relating to laying-out, which we largely abstract away at this point). ERA is intentionally applicable to a range of domains of interest for generative methods, and is growing in popularity and acceptance. There is a virtuous cycle whereby the more the approach is used, the more useful it become due to increased availability of points of comparison.

## 6.2 Dungeon metrics

In this section we discuss the (non-)applicability of the linearity and leniency metrics proposed in the preceding section to the domain of dungeon graphs. We note the contributions of Lavender and Thompson (2015) to this domain, and observe as they did the differences between map and mission linearity. We observe that our selection of comparable assumptions in Sec. 5.2.1 has facilitated eventual direct comparison between evaluation outputs. We reproduce a sample output from Sec. 5.2.4 to facilitate explanation of the metric definitions, and briefly describe it. Then we fully explain the formal metric definitions, and present values for those metrics for our exemplar output in Fig. 6-1. Finally we present a fuller description of the system developed by Lavender, with specific reference to the variant and control grammars, and the shared assumptions that we have made.

It is important to note that we calculate the evaluation metrics directly from the dungeon structure graph - this method assumes that no significant changes to the dungeon will be made during the process of laying out the dungeon as a playable level. This assumption holds in the

current system but is not a necessary technical limitation.

### 6.2.1 Metric definitions

We use the following four metrics, as defined by Lavender (2016) and based on the original Linearity and Leniency metrics proposed by Smith and Whitehead (2010):

**Mission Linearity:** the number of nodes on the shortest direct path between start and end of the mission graph, divided by nodes within the graph total.

In Fig. 6-1 this is $9 \div 19 = 0.473684211$.

$$missionLinearity = \frac{Number\ of\ Nodes\ on\ Shortest\ Path}{Total\ Nodes\ in\ Graph}$$

**Map Linearity:** a weighted scoring of each room with one or more forward exits divided by all rooms with any forward exits: those with a single entrance and exit (fully linear) have weight 1; those with two forward exits have weight 0.5, and those with three exits are considered maximally non-linear and do not contribute to the numerator. 'Dead ends' (rooms with an entrance but no forward exit) are not directly counted by this metric.

In Fig. 6-1 this is $(1 \times 12 + 0.5 \times 3) \div 15 = 0.9$.

$$mapLinearity = \frac{(1 \times SingleExits) + (0.5 \times DoubleExits) + (0 \times TripleExits)}{Total\ Rooms\ with\ Exits}$$

**Leniency:** the proportion of safe rooms within the dungeon graph to total rooms. For the purposes of this evaluation we have considered only local combat challenges and the final Boss node to be 'unsafe', though the precise calculation of this metric is to a degree dependent on the details of the final realisation of a level: it is possible that any of the traversal or puzzle challenges or even dungeon item locks could be implemented in a way that was potentially 'unsafe' for the player character.

In Fig. 6-1, this is $15 \div 19 = 0.789473684$.

$$leniency = \frac{Number\ of\ Safe\ Rooms}{Total\ Rooms}$$

**Path Redundancy:** the number of rooms that are present but do not need to be visited in order to complete the level, divided by all rooms. In (Lavender, 2016) these are defined as rooms that "do not eventually lead to, or themselves contain, any reward", and are byproducts of possible expansions of the graph-rewriting rules used in that system. However under the ASP formulation described in Sec. 5.2.3 these rooms are only generated as optional '`exploration`' paths leading to non-critical `reward` nodes — a comparable but not identical concept.

In Fig. 6-1, these are the combat challenge and dungeon item lock blocking access to the Map optional reward, and the measure is $3 \div 19 = 0.157894737$.

$$redundancy = \frac{Number\ of\ Non\text{-}critical\ Rooms}{Total\ Rooms}$$

### 6.2.2 The Zelda Dungeon Generator

The approach in Lavender (2016) uses a pair of grammars to generate a mission graph and, from it, a mission space. Several variants of the grammars are described, and ERA applied to visualise the changes that result in the output space; some of these visualisations are presented in Fig. 6-3 for comparison to our work.

## 6.3 Expressivity analysis

Having selected appropriate metrics according to the approach laid out in Smith and Whitehead (2010) and definitions provided by Lavender (2016), we generated 1,000 dungeon graphs following the approach in Chapter 5, and for each graph calculated the value of the four general metrics. The outcome of this approach is visualised in Fig. 6-2, where the colour of each square bin in the plot corresponds to the quantity of dungeon graphs possessing those metric values. The expressive range analysis consists of four main steps (Smith and Whitehead, 2010):

- Determine appropriate metrics. As we intend to contrast the outcome of this analysis with existing visualisations, we will use the same metrics, as defined below. In addition, we measure and report the average size of generated graphs.
- Generate content. We collect 1,000 individual sample dungeon models from separate seeds and collate the metrics scores for each.
- Visualise the generative space. We use matplotlib to render heatmaps of pairs of metrics (Fig. 6-2), comparable to the existing visualisation by Lavender (2016) (Fig. 6-3).
- Analyse the impact of parameters. In Sec. 6.4 and Fig. 6-5 we compare and contrast the effects of slight alterations to the problem formulation.

Shown in Fig. 6-2: three views of the expressive range of the full ASP formulation (Sec. 5.2.3). This approach to visualisation reveals several potential weaknesses of that formulation that were not directly apparent from the individual specimen inspection performed in Chapter 5. Notably, the tight clustering on the Leniency and Map Linearity axes indicates a lack of possible variety in the values of these metrics across all sampled outputs (i.e. generally highly lenient levels, highly linear maps). While it may in fact be desirable for outputs to cluster near these specific values for certain contexts, we wish to show that this generation approach is capable of a broader expressive range. A small number of informed changes to the ASP formulation (Sec. 6.4) results in the considerably more varied output shown in Fig. 6-5.

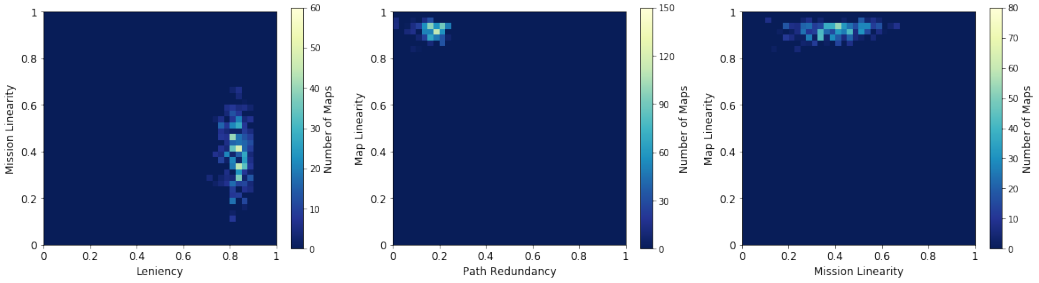



Figure 6-2: Three views of the expressive range of the initial ASP formulation (Sec. 5.2.3).

Several distinct behaviours of the generator are clear from the visualised heatmap. In general, levels are fairly strongly clustered around a few specific areas, indicating a lack of variety between generator outputs. The leftmost plot shows that all sampled levels are highly lenient, likely due to the low theoretical maximum proportion of dangerous nodes (the maximum possible is five local combat challenges according to the bounds in Table 5.3, plus one `boss` node, totalling 6, and the average graph size was 22.363). The second plot shows low path redundancy and high map linearity, likely due to a combination of the rule that forbids dead-ends that don't provide rewards (Listing 5.4) and the rule that constrains the number of exploration nodes to 5 or less (Listing 5.5). There are also notable gaps despite the clustering: due to the enforced variety of local challenges and the guaranteed presence of a Boss node, it is impossible for any graph to reach the theoretical maximum Leniency value. Similarly, due to the requirement that the `BossKey` must be terminal (Listing 5.3), it is not possible for any map to be fully linear.

### 6.3.1 Comparison with the Zelda Dungeon Generator

In this section we present a number of comparisons and observations relating to the differences in ERA outputs between our work and those of the the Zelda Dungeon Generator (ZDG). As in Horn et al. (2014) the intent is not to prove unambiguously that any given approach is better; rather to provide a grounding over which it is possible to reason about the features and affordances of different approaches — the choice of approach will frequently be contextual based on a range of factors, but provision of a visualisation of the output space can help to inform that decision. Several of the implementation assumptions that we have made have been informed by both the domain of Zelda dungeons generally and also some of the restrictions assumed by the ZDG in order to facilitate direct like-for-like comparison of this nature.

Fig. 6-3 reproduces three of the outputs of the expressive range analysis performed by Lavender on the system detailed in Lavender (2016) and summarised in Sec. 6.2.2 — specifically, the outputs relating to the Control rules: a set of graph- and space-rewrite rules based on those in Dormans (2010). These rules are intended to provide a balance between the other deliberately biased rulesets analysed in that work, and therefore are the most representative point of comparison. The heatmaps reveal a good, central spread of values for leniency and mission linearity, but incredibly tight clustering on the path redundancy metric, apparently

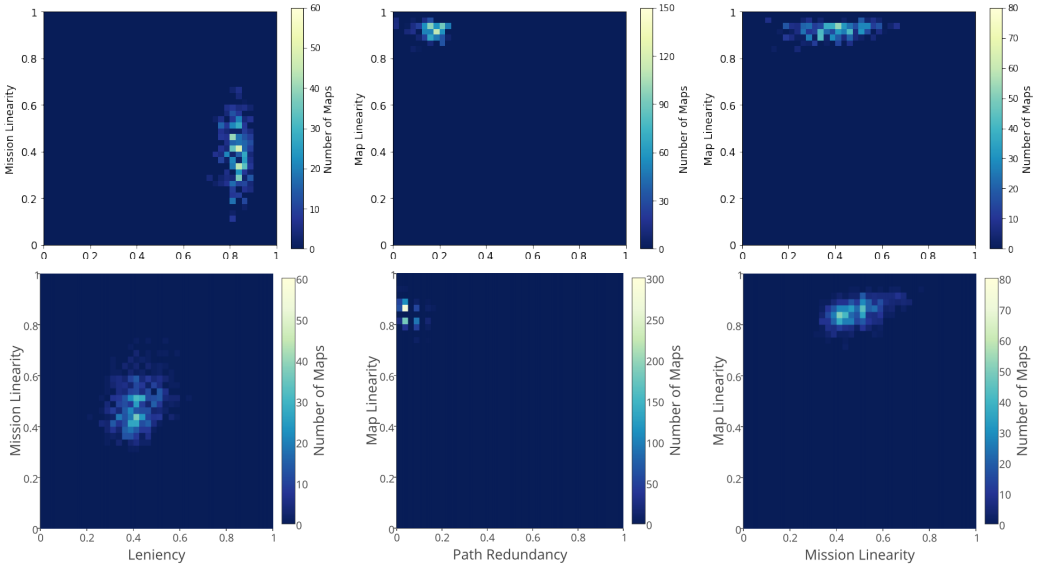


Figure 6-3: Figures 61, 67 and 70 (below) from Lavender (2016), representing evaluation of Control Rules. Reproduced with permission. Compare with Fig. 6-2, reproduced above.

due to limitations of the mission graph used. Between the two approaches, the spread of values for both mission and map linearity are reasonably similar, with the primary difference across all four measures being the extreme comparative leniency of the ASP-based levels.

The approach in Lavender (2016) uses a pair of grammars to generate a mission graph and, from it, a mission space. The Mission Linearity and Leniency view (left) is evaluated over the output of the Control mission graph grammar; the Map Linearity and Path Redundancy view (centre) is evaluated over the output of the Control shape grammar over a single mission graph ((Lavender, 2016, Fig. 66, p. 78), not reproduced here) and the Map Linearity and Mission Linearity view (right) is evaluated over the output of the Control shape grammar over the output of the Control mission graph grammar. These differing sources explain the gap in Map Linearity at about 0.85 in the centre view, which Lavender suggests may be an artefact of the size of mission chosen.

## 6.4 Investigation and alterations

In this section we pick up on some of the critiques of the output presented in sections 6.3 and 6.3.1, and suggest concrete alterations to the formulation that could address them. We suggest the expected impact of these alterations, then present an updated ERA including corner plot to show the actual outcomes. We conclude with a three-part side-by-side comparison of output from the initial formulation, Lavender's ZDG, and the updated formulation.

Motivated by the observed clustering in the original visualisation, we investigate the impact of three minor changes to the problem formulation. Working under the hypothesis that the initial path redundancy and map linearity clustering were due to the dead-end and exploration restrictions (Listings 5.4 and 5.5), we weaken the former from "`:- terminal(N), not`

| line | Initial | Altered |
|---|---|---|
| 2 | `2 { node(p(1..5)) } 5.` | `2 { node(c(6..10)) } 5.` |
| 31 | `:- terminal(N), not rewd(N).` | `:- rewd(N), not terminal(N).` |
| 43 | `:- 5 {exploration(N) : node(N) }.` | `:- {exploration(N) : node(N) } 5.` |

Table 6.1: Three informed changes to the dungeon generation formulation.

`rewd(N).`" to "`:- rewd(N), not terminal(N).`"[2], and we invert the exploration constraint to require a minimum of 5 exploration nodes, rather than 5 maximum (see Table 6.1). The effects of these changes are clearly visible through comparison of Figs. 6-2 and 6-5 — a considerably broader spread. The third change was to replace all potential puzzle nodes with potential additional combat; resulting in a small but notable decrease in the general leniency. Fig. 6-4 illustrates a sample dungeon graph generated under the new rules, and clearly shows the effects of constraints requiring increased exploration nodes. As with Fig. 5-7 in Sec. 5.2.4, a single specimen can not necessarily be expected to be representative of the typical output, and so Fig. 6-6 provides a thorough visualisation of the new space.

Fig. 6-5 shows three views of the expressive range of the altered output generated via ASP after making the changes listed in Table 6.1. Note that the variance in Path Redundancy and Map Linearity values has greatly increased compared to the original formulation shown in Fig. 6-2; the average leniency has decreased; there is increased clustering around the theoretical minimum values for Mission Linearity and Path Redundancy; and the minimum possible value for Path Redundancy appears to be higher. These changes are as expected based on the alterations in Sec. 6.4; informed by this new visualisation additional changes could be made in order to attempt target certain areas of the possible expressive domain if desired, or further broaden the variance.

Fig. 6-6 presents a corner plot showing each of the combinatorial views of the the output data visualisations after making the changes described in Table 6.1. Though three of the views duplicate those in Fig. 6-5 presented for comparison purposes, the remaining three and the single-metric histograms provide additional detail regarding individual distributions. From the Path Redundancy histogram it is apparent that the sampled outputs do not vary smoothly over that metric but rather cluster at a range of specific values, whilst the Leniency variable, viewed in isolation, reveals a continued tight clustering that is more pronounced than is apparent in any of the 2D plots. Visualisations of this nature can help to guide informed changes to the problem formulation, or reveal previously hidden flaws or inexpressible areas within the expressive range (Smith and Whitehead, 2010; Lavender, 2016).

Visibility is key to informed development and refinement. Application of ERA allows for fluid exploration of the space via surfacing the effects of any changes.

[2] 'forbid terminal nodes that are not rewards'→'forbid rewards that are not terminal'

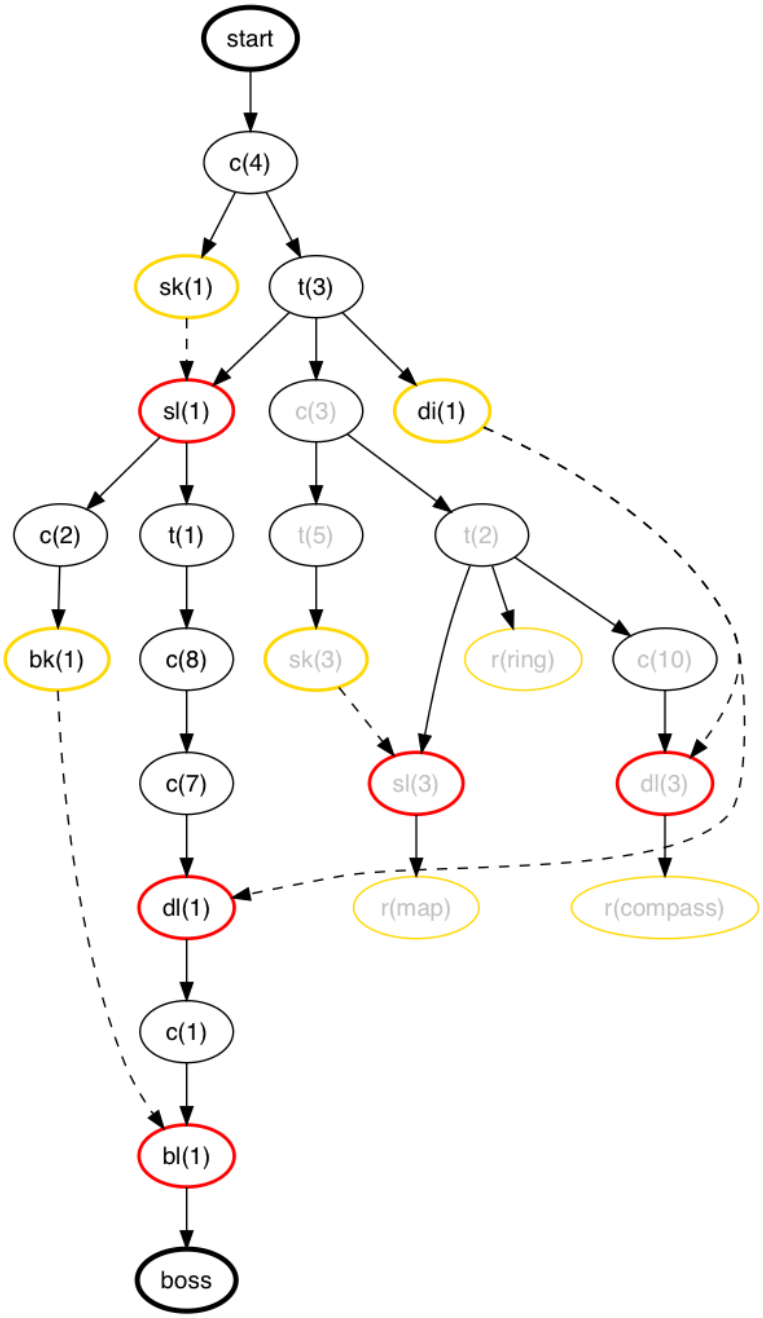



Figure 6-4: A sample map generated after the changes detailed in Sec. 6.4, showing additional redundancy and nonlinearity.

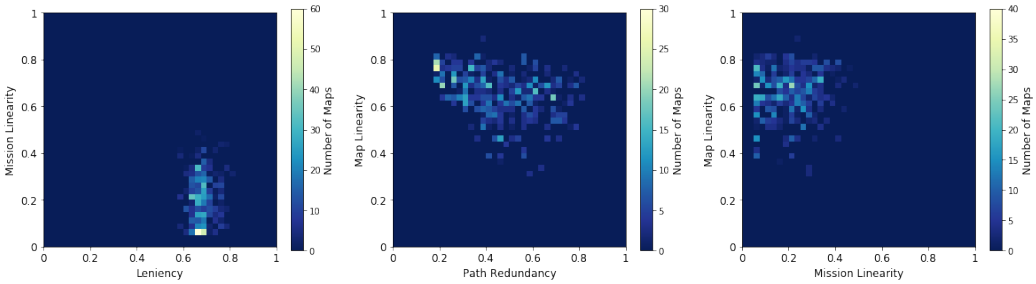



Figure 6-5: Three views of the expressive range of the altered output generated via ASP after making the changes listed in Sec. 6.4.

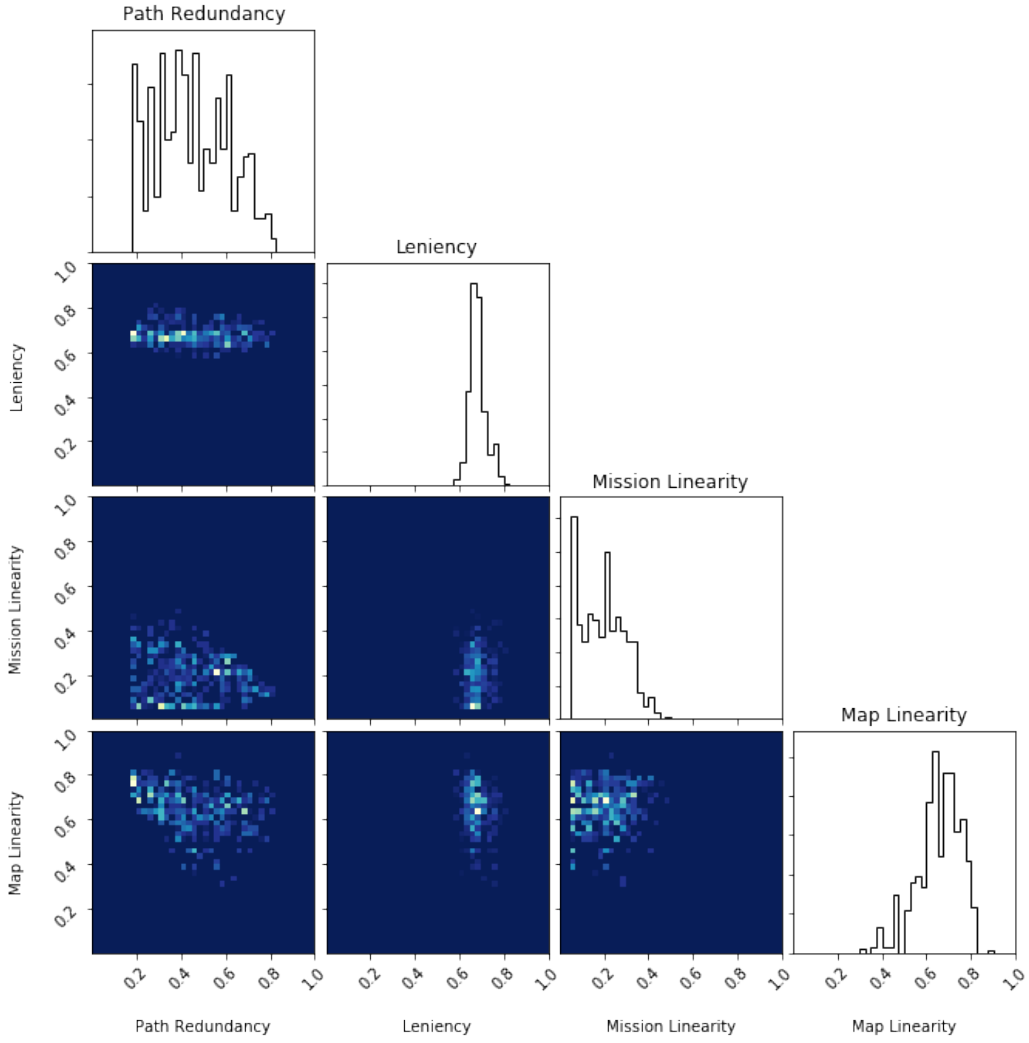
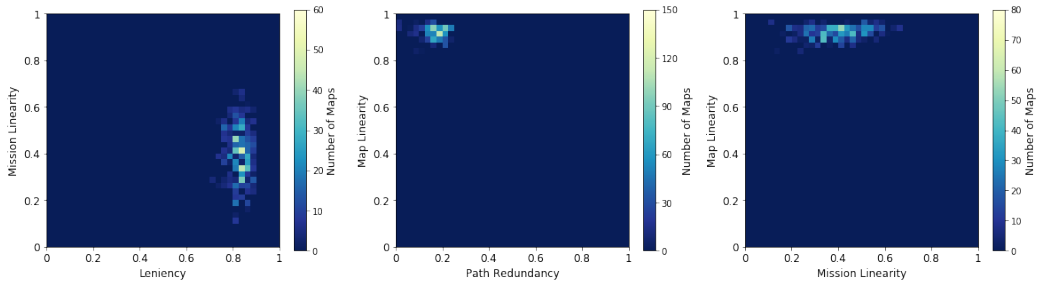
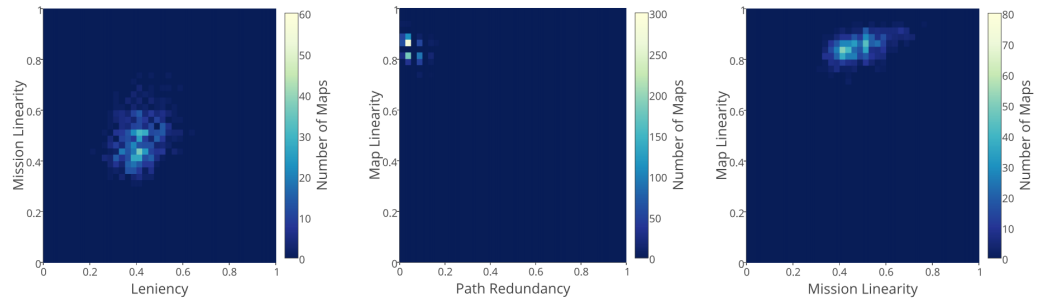


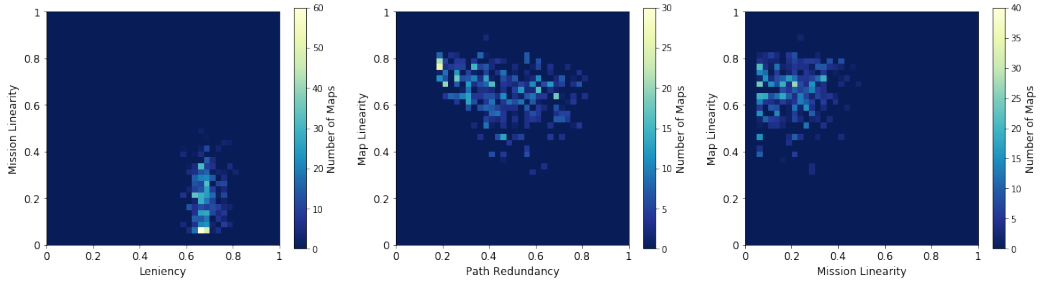

Figure 6-6: A corner plot (Foreman-Mackey, 2016) showing each of the combinatorial views of the the output data visualisations after making the changes described in Sec. 6.4.




(a) Initial formulation (Sec. 5.2.3).




(b) ERA output from Lavender's Control rules (Sec. 6.2.2).




(c) Updated formulation (Sec. 6.4).

Figure 6-7: A three-way comparison of the initial formulation described in Chapter 5, the output analysis of Lavender (2016), and the altered formulation (Table 6.1).

### 6.4.1 Domain alterations

In this section we discuss the flexibility of the ASP approach by reshaping the formulation to more closely match the assumptions of more-linear action-adventure levels *without* a 'spatial puzzle quality' — i.e. more like the levels in Ninja Theory's DmC/Disney Infinity than the ones in Hellblade (or, indeed, Zelda games).

| line | Initial | Altered |
|---|---|---|
| x | `2 { node(c(1..5)) } 5.` | `more combat.` |
| x | `2 { node(t(1..5)) } 5.` | `more traversal.` |
| x | `no dungeon item` | `none` |
| x | `no boss key and lock` | `none` |
| x | `all locks on main` | `path to boss` |

Table 6.2: Fomulation changes to produce more-linear dungeons.

Answer Set Programming (ASP) is less necessary for this kind of domain as the domain-specific constraints are greatly relaxed without the possibility for complex spatiotemporal relations between keys and locks. However, this is a common domain in games development, and showing that with minor alterations a tool intended for dungeon generation can also generalise to an adjacent domain is of use and interest — e.g. Mario level generation[3].

## 6.5 Discussion

The dungeon generation field contains a number of disparate approaches to generation, each with distinct benefits and drawbacks. It has traditionally difficult to draw direct comparisons between multiple generators (in the manner of Horn et al. (2014) for the Mario domain) — in part due to each generator working within a subtly distinct subdomain, but also because even for approaches within the same domain it's often not clear from the inspection of a few specimen outputs how two generators differ/match. Two known extant solutions: the Generative Design in Minecraft Competition (GDMC), and the work by Horn et al. (2014). The former is a special case; multiple different generators run in deliberately identical contexts, and evaluated by experts. The latter signals a more feasible approach for future widespread adoption: publication by generator authors of ERA evaluations of their own work, for later comparison by other authors in the same domain. For this theis, we build on and are appreciative of the work by Lavender and Thompson (2015), and have made available our own expressive range analyses in turn for future comparison.

In addition to the four domain-general metrics used, there are a number of additional metrics that may be relevant to end-users of the system, if not directly for the ERA process itself, such as average size for a given encoding and configuration.

[3]a number of new aspects would be relevant and necessary for that domain, though some of the concepts (repetition and elaboration) would be useful to develop a representation for within the ASP fragment repertoire, to facilitate backporting to the dungeon domain and investigation of how they could be applied...

**Average Size:** the sum of all nodes in all generated graphs, divided by the number of graphs generated. In addition to the four general metrics listed above, we note that changes to the generation formulation can affect the average size of the generated graphs — a measure which each of the other metrics relate to, in some degree. In Fig. 6-1, there are 19 nodes.

$$avgSize = \frac{\sum \textit{ Total Nodes in Graph}}{\textit{Total Graphs Sampled}}$$

Whitehead (2020) suggest a new domain-independent metric relating the critical path to individual opportunities for choice. Though this work is not (publicly) applied to dungeon levels, the metric could productively be applied and recent publication shows ongoing interest in the area.

There is also potential for a more in-depth comparison of the generative space characteristics of a wider range of generators within this domain, along with investigation into altering parameters, grammars or constraint formulations. A study by Horn et al. (2014) in the domain of 2D platformer (Mario) levels considered seven generators from literature and levels from the original Super Mario Bros game. There are a comparable number of dungeon generators in literature (Dormans, 2011; Lavender, 2016; Karavolos, Bouwer and Bidarra, 2015; Smith and Bryson, 2014; Baldwin et al., 2017; Heijne and Bakkes, 2017; van der Linden, Lopes and Bidarra, 2013; Valtchanov and Brown, 2012, present paper), though this domain lacks the broad consensus on common assumptions and definitions the domain of Mario generators has achieved.

# Chapter 7

# Conclusions and Future Work

In this chapter we summarise the thesis so far. We review what has already been presented, and motivate an argument that these represent a grounded, original, significant contribution to the field. We explicitly summarise the three main contributions presented in Chapters 4, 5 & 6, noting publications at FDG and STAIRS, and citations by TAKSIM at CoG'19. Then, we briefly review a range of directions we have identified as promising for further study in the individual chapters, and signpost some larger areas: We cover plans made for further (qualitative) analysis of the system presented in Chapter 5; we discuss promising indicators for application of these techniques to other domains; and we present a background on the initial intended direction of the project overall: constraint as a generative approach for populating semantic ontologies, including sample literature. Finally, we conclude with a reiteration of our contributions and how we believe they present a contribution to the field.

To demonstrate the contribution of the research, it is important to show the that the work is justified, thorough and sufficient. In Chapter 2 we surveyed a selection of existing academic research that covers both the philosophical aims and major goals of the field of Procedural Content Generation (PCG), and also many of the major approaches detailed in literature. Chapter 3 presented a more detailed investigation of Answer Set Programming (ASP); a technique that has recently shown promise as applied to a range of varied generation problems that we also detail. We present our own initial application of ASP to the industry-relevant problem of combat wave progression generation in Chapter 4, and discuss the implications on designer workflow and higher-level design decisions. A more complex industry-driven problem is detailed in Chapter 5, where we investigate the application of ASP to modelling the generation of dungeon-style levels in the service of providing an explorable and controllable level greybox generator for the commercial engine toolset UE4. Chapter 6 then presents our application of an academically-popular generator evaluation approach (Expressive Range Analysis (ERA)) to the generator detailed in Chap. 5, and comparison of our results to the output of another generator in the same domain that uses a contrasting generation technique; the the Zelda Dungeon Generator (ZDG). Finally, in this chapter we next summarise the contributions to the field presented in the thesis, then review the opportunities for additional development

that have been raised by previous chapters, introduce some more significant implications for further work, and conclude with a summary of the work.

As an overall evaluation of the approach: we must demonstrate generality of the research by successful application to a range of problems. We have shown two that are specifically relevant to our industry partner, plus contributed to a growing body of ERA application datapoints. We further show quantitative evaluation demonstrating comparability with other work in the domain, and discuss possibility for evaluation by game design experts. For future development of the approach it seems reasonable to attempt problems for which there exist current reference generators for comparison — either in popular domains (infinite Mario) or as part of generation competition (procedural Spelunky, GVG-AI level generation track, Starcraft map generation, or the GMDC).

## 7.1 Summary of contributions

In this section, we recap the main contributions of Chapters 4, 5 and 6.

Chapter 4: Wave-Based Combat: changes in design could require rebuilding progression. We have achieved: automatic, varied generation including hand-placed landmark waves, using an approach that is easily alterable and regenerable in the presence of changes. This opens many possibilities: for designers ease-of-use, live/dynamic generation; for further development: skill-teaching

Chapter 5: Contrained Dungeon Design: application of ASP to a more complex problem, demonstration of a viable alternative to existing approaches in literature. In addition: integration with industry software and illustration of intended use case. The problem: early 'sketches' of action-adventure levels can contain complex structural constraints that are not explicitly represented within available editing tools. We have achieved: a controllable system for exploring a space of possible action-adventure levels that includes explicitly-represented key-and-lock puzzles, guaranteed connectivity from start to finish, and annotation for optional exploration and reward routes and varied local challenges. There is comparability to existing work within similar domains.We have provided an initial implementation of casual-creator style exploration of the relevant greybox space, and further work in the field builds on ours (TAKSIM; Abuzuraiq, Ferguson and Pasquier (2019)).

Chapter 6: Expressive Range Analysis: application of an evaluation approach with increasing recognition within the field, and use of this approach to perform a direct comparison with previous work on the same domain. The problem: by individual inspection of sample outputs from a generative system, it is difficult to get an idea of the broader scope of what it can or does produce, or compare it to other generators in the same domain. Using an existing evaluation approach, we investigate the output of the system in Chap. 5, and are able to observe characteristics of the output that are not immediately obvious from the formulation alone. After comparison with an existing generative system that has also used ERA, we are able to make a small number of informed changes to the formulation, whose impact we investigate through new analysis.

## 7.2 Future work

In this section we review suggestions made in Chapters 4, 5 and 6 for how the presented systems could be further developed, and present some additional links between the chapters that arise naturally from the progression of the research (e.g. ERA for Wave-Based Combat (WBC)).

Chapter 4: Wave-Based Combat suggested a range of further possibilities, developing on the work presented: experimentation with more complex, semantic-driven constraints (`swarm`, `solo` etc.), more sophisticated composition of difficulties, true mixed-initiative exploration including preserving good content, a more usable interactive interface than direct ASP editing, and approaches for evaluation including ERA — despite scarcity of comparable data. The most productive avenue for further research would be skills-based progressions including abilities and n-grams.

In Chapter 5: Contrained Dungeon Design: possible alternative generation formulations including cycles, less constrained reification, true mixed-initiative exploration including preserving good content and recycling facts emitted from in-engine alterations, automated visualisations to facilitate exploration (both Boss Key and ERA). The most productive avenue for further research would be complete level generation — including lower-level population of progressive enemies, challenges and puzzles, and higher-level considerations such as foreshadowing and layout symmetry.

In Chapter 6: Expressive Range Analysis we suggest investigation of the utility of broader metrics for the dungeon domain, or application of the dungeon metrics to a larger number of dungeon generators.

We pick up on the question of evaluation to address a weakness of ERA: though it can quantitatively address the question of the breadth and direction of content produced by a system, without human evaluation it is difficult to qualitatively answer questions about the applicability and value of what is produced.

### 7.2.1 Plugin user study

In this section we present the motivated need for a user study to provide qualitative analysis of the system presented in Chap. 5, via the plugin and engine integration. We discuss key functional aspects and learnings from a recent paper on another study on a mixed-initiative generative system Alvarez et al. (2018), lay out our own schedule and plans for a similar study, and briefly describe the documents presented in the appendices.

There is a desire for qualitative analysis of usability by expert users, and a possibility for analysis by non-expert users and comparison between outcomes. Two main axes of familiarity would be relevant to these concerns: with playing/designing dungeon levels, and with the Unreal Engine 4 (UE4) interface.

A previous study in a similar domain (user-driven content generation) is presented by Alvarez et al. (2018), and informed this design. Personal discussion with the authors revealed that as some study participants were less familiar with the dungeon concept they might have benefited from an opportunity to experience a generated dungeon from the player's perspective,

to prime expectations. Discussion also suggested the benefits of simultaneously recording the tool screen, audio description of actions, and user verbal and facial responses.

Quote from data plan: "A software tool has been produced to aid in the development of prototype 3D environments for games, subject to gameplay/designerly constraints. Data will be collected as part of a user study investigating usage of the tool for a small number of typical scenarios: after a briefing and preliminary questionnaire to elicit relevant experience(s), users will be recorded whilst undertaking a brief tutorial and a small number of tasks using the tool. Participants will be encouraged to speak aloud about their intentions and impressions, and on completion of the interactive portion of the study will be debriefed and given an opportunity to share further thoughts or ask any remaining questions."

The session structure consists of: (i) an initial study briefing, review of the Participant Information Sheet (attached), opportunity for questions, Consent form (attached) ( 5mins); (ii) tutorial and experience elicitation ( 10mins); (iii) series of structured generation tasks, opportunity to experiment with the software (time permitting) (25-35 mins); (iv) debrief and final questions (5-10mins)'

For more detail see the original data plan, participant consent form and participant data sheet attached in the Appendices.

### 7.2.2 Further domains

In this section we present a range of additional domains that we have identified as potentially suitable for the application of similar techniques, alongside brief discussion of applicable literature, and intended methods for evaluation including by comparison to specific leading alternative generators for those domains.

A number of further domains are potentially amenable to ASP-driven generation according to the discussed criteria — more than could feasibly be addressed during the the project. A non-exhaustive selection are presented here, with reference to the particular benefits each potentially affords.

Procedural platform-game level generation is a popular area for academic generation and evaluation research. It is also a highly structured domain — ASP is potentially well-suited to performing grammar-like construction of this type of content, though it is likely to be necessary to implement the discussed iterative refinement approach in order to manage the scope of individual reasoning tasks. Development of an analogue to the grammar-based generator in Dahlskog, Togelius and Nelson (2014) would allow rich comparisons with a range of existing academic implementations, and potential for competitive evaluation.

Another project for which the system may be well-suited is the generation of procedural open play spaces from pre-existing parts using an approach similar to Smith and Bryson (2014). In several modern games, the majority of the environmental assets are designed as 'kit parts' that may be combined and re-used and arranged in a variety of configurations. Given appropriate semantic annotations there is potential for these kit parts to be used with an ASP-based layout solver similar to the one described in Tutenel et al. (2009a), in order to automatically populate

a space with correctly arranged components. In combination with the initial proof-of-concept work done on producing rogue-like dungeon spaces, this could become a multi-step generation process as laid out by Smith and Bryson (2014), where the generator creates a range of connected gameplay areas and then fills them with appropriate content. Alternatively, either step could be used in isolation to augment a human designer's workflow: the system could provide an initial layout for an area, or automatically populate a suitably-annotated layout specified by the designer. As with the combat wave progression generation project, there are no obvious existing analogous generators with which to perform a direct comparative evaluation. Aside from human qualitative evaluation by expert users and potentially end-users of the produced content, the most appropriate evaluation approach will be to follow the recommendations in Smith and Whitehead (2010) for selecting relevant domain metrics.

A final area of potential interest is the domain of narrative generation. This would be reasonably distinct from other considered content types, as the output would be an intangible overall structure for story, plot-points and individual missions or scenes. Output from the system could be used as input for other generator systems for characters and locations. Existing literature suggests implementation of a pre-requisite/theme-based system as described in Carmichael and Mould (2014), which would require further work on an appropriate node-based interface for visualising and manipulating abstract concepts. It would be sensible to approach this after other aspects of the system are more fully developed.

### 7.2.3 Semantics and OWL

Formal representation of a domain of interest may also be useful for producing games, as well as enabling better discussion of them. An active area in games development research is the application of semantic data to improve consistency between appearance and affordance in virtual worlds. For PCG, this involves establishing a layer of 'meaning' meta-data about objects and features in the game world, and how they might relate to one another. A designer producing part of a 3D world might understand that the player's path through a level must necessarily pass through a particular door, that the door swings open through a particular volume, and therefore that that volume must not become blocked by other elements of the level design if the level is to remain completable. capture and record designers' intentions explicitly. This ontological description of game world elements can then be used to produce constraints on generation, in order to ensure that generated content makes sense according to expectations players may have based on its visual appearance.

Development of a content-agnostic generator system as described would approach the goal of being able to produce 'plug-and-play' PCG middleware for a wide range of games and content types Togelius et al. (2013a).

Semantics is the study of meaning — applied to language, it relates to the meaning of a statement (in contrast to syntax, which relates to structure). Applied to virtual environments, it relates to information conveying the meaning of an object (Tutenel et al., 2008), potentially going beyond basic geometry to specify physical properties, roles, behaviour, etc. Often, the

semantics of a word or object are defined in relation to a specific domain of interest, which may be defined in an ontology[1]: a formal categorisation of the types, properties and relationships between entities in the domain. Ontologies are a knowledge representation technique that allow specification of and reasoning about the relationships between entities in the domain of interest. They are an active area of research in both AI and the Semantic Web (see Sec. 7.2.3), as they support description of large domains of potentially incomplete information, and provide the ability to infer the existence of relationships such as class membership or 'same-as' relations between individuals.

Research into formal semantic representations has often been concerned with appropriate organisational structures for information, to best represent and describe real-world domains. As an example, Zagal et al. (2007) describe initial work towards a 'Game Ontology' — a unified vocabulary for describing games and their constituent parts. They identify the important structural or design elements that are either common to or distinct between a wide range of concrete examples in an attempt to describe the design space of games. The intention is not to provide a taxonomy of all games, but to specify a range of important compositional elements along with representative examples. Their hierarchical approach considers entries relating to *interface*, *rules*, *goals*, *entities* and *entity manipulation*, and abstracts away from issues of setting or content in favour of specifying only game design related information. In later work on the Game Ontology Project (Zagal and Bruckman, 2008) they present the information in the form of an editable wiki and encourage contribution in order to develop the vocabulary of concepts relating to games design.

An early general overview of the role of semantics in games is given by Tutenel et al. (2008), who suggest that "when semantic data is linked to procedural generation techniques, the power, quality, and realism of these techniques can be improved". They distinguish three distinct 'levels' of semantic information: *object semantics*, *object relationships* and *world semantics*, and give illustrative examples of each. Constraint solvers are highlighted as a possible approach to maintaining semantic consistency, by expressing relationships between objects in the form of constraints. They discuss a range of constraints that may be appropriate, and detail existing systems that use either general constraint-based approaches or domain specific solvers to generate content. They claim that lifelike behaviour increases realism of virtual worlds, leading to immersive and compelling gameplay, and conclude that there are numerous promising applications for semantic infomation in virtual worlds. Three outstanding issues are noted: (i) the need for a domain-general and explicit approach for defining and specifying semantic data, (ii) the need for powerful constraint-solving methods to maintain the consistency of relationships between evolving objects in a changing environment, and (iii) the need for better integration of semantic data with procedural generation techniques.

To demonstrate a specific application of semantic information to improve the design process portion of content production, Tutenel et al. (2009b) present a system able to generate convincing 3D residential interiors, and discuss the added value of considering semantics during the

[1] 'Ontology' is the study of the categories of being and their relations; 'an ontology' is a formal representation as above.

design phase. They develop a tightly-integrated semantic class library and rule-based layout solver that is able to derive constraints from the relations between classes in the library, and then place objects into the scene according to these constraints, backtracking where necessary to preserve validity of the layout. Classes in the library are defined according to an inheritance hierarchy that specifies common properties, including height, width, existence of internal storage space or clearance requirements, and also associates the class with a number of rules for spatial relationships with other objects. Initial generation is guided by designer-provided additional constraints on the output, such as a desired total storage volume, quotas for specific classes of objects, or constraints on maximum age of objects to specify modern items only. The system is also able to verify designer item placements according to the included rules, and continue to generate additional items accordingly, using a custom layout solver detailed further in Tutenel et al. (2009a). They conclude that both manual and procedural game world design can be improved by capturing and using additional information about designers' intent in the form of semantics of game objects. The primary benefit they state is the potential for automation of parts of the design process, though they note that availability of additional semantic information may also create opportunities for improvements in other areas, such as improved audio-visual effects or interesting AI interactions.

Tutenel et al. (2011) suggest the use of procedural filters for customisation of game worlds, proposing that editors can make use of semantic knowledge for improved filters.

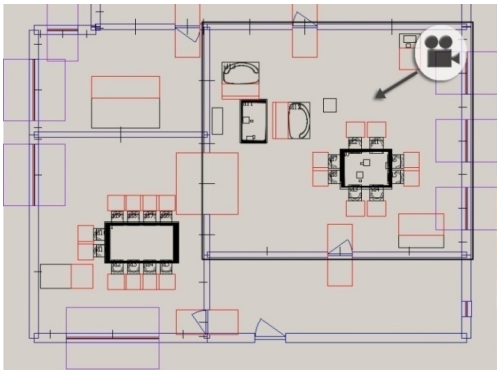


Figure 7-1: 2D floorplan for a generated environment, showing clearance areas around furniture and doors (from Tutenel et al., 2009b).

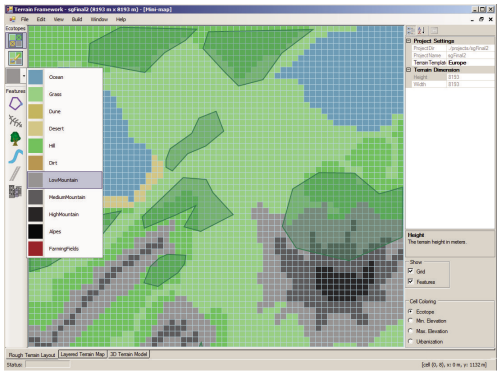


Figure 7-2: A sample scenario terrain sketch. Cell colour defines ecotype, green polygons are vegetation (from Smelik et al., 2010).

Smelik et al. (2010) describe a simple, declarative system for sketching terrain elements, which are then converted to realistic 3D terrains for military training exercises. They note that producing appropriate content manually is often difficult to do at scale due to the complexity of the many interactions between features that must be considered by the designer. They suggest that if a portion of that knowledge is transferred to the generation system – via some form of semantic constraints – this allows the use of simpler, declarative models for content production. In their proposed system, local corrections could be made automatically, including flattening

terrain beneath buildings, carving road embankments in mountains, or placing bridges where roads cross water. In the implemented system, semantic consequences are represented on an ad-hoc basis, for example certain kinds of plant will only grow where elevation and water density are within acceptable bounds. This allows for automated placement of individual plants of the correct type for the region, within areas that according to the sketch should be covered in vegetation (see 7-2). They conclude that a declarative approach to terrain construction could support and enhance the modelling process for military exercises, but note consistency management between many interacting terrain features is likely to necessitate the use of constraint solving methods.

It should be possible to annotate existing objects and compose or select design constraints with as little friction as possible in order to encourage experimentation and rapid iteration. One possibility would be to leverage existing knowledge representation techniques developed in other areas such as the Semantic Web.

There is already a significant body of existing research on semantic knowledge representation in areas relating to the Semantic Web – the effort to mark-up information online to make it easy to process and reason about automatically. One of the key contributions is OWL, the Web Ontology Language. OWL is a format that allows explicit defintion and specification of semantic data in the form of classes, entities, properties and the relationships between them (Antoniou and Van Harmelen, 2004), There are also a range of reasoners available for OWL and its various sub-languages, that support inference tasks relating to class membership, consistency checking, etc. However it appears that semantic data is almost exclusively used for the description of existing entities rather than the generation of new valid-but-fictional information within the described domain.

As with ASP (Sec. 3.2), there is a recognised need for graphical editing tools to simplify the process of developing and refining OWL content. Protégé is a free, open source grapical ontology editor developed in Java, which is able to interface with various reasoners and save OWL ontologies in a range of possible formats (Knublauch, Musen and Rector, 2004)[2]. A graph-based editor for ontologies and rule bases is presented in Bak, Nowak and Jedrzejek (2013), and there are also a range of libraries, utilities and plugins available in a variety of languages for processing and working with OWL (see Fig. 7-3).

The `dlplugin` for `dlvhex` handles querying ontologies and translating the response into ASP during reasoning, potentially with additional information derived from the existing partial solution. No similar system currently exists for Clingo, however for the purposes of this project it should be sufficient to simply query and translate an ontology as a pre-processing step, using an approach similar to the one set out by Gaggl, Schweizer and Rudolph (2015). If it is ever necessary to make use of partial solutions to augment the query, this may be approximated using Clingo's multi-shot solving capabilities.

The combination of ASP and semantic data is likely to be appropriate and useful for large domains with discrete structure and constraints on validity – e.g. while it would be possible to approximate a basic Markov name generator using this system, simpler approaches would

[2] `https://protege.stanford.edu/` — accessed 27 November 2020

also be sufficient and potentially more efficient. Fortunately, aside from procedural racetrack or noise-based heightmap generation, many areas of interest for generation in games consist of discrete, structured data under interesting constraints, as detailed below. Finally, the potential of a given domain for effective evaluation will also be a relevant concern. Some domains, such as platform game level generation, have a large body of existing literature to draw guidance from and comparisons with. Other domains provide scope for competitive evaluation via entry to an academic competition for comparative assessment, as covered in Sec. 2.5.

Further work currently includes defining a broader base ontology for a range of common cases, and improving solver support for these tasks. The system needs to support writing data back into the ontology for iterative refinement, and reading fixed facts for mixed-initiative generation. A transformation scheme would guide progression between tasks, and further consideration is needed for the method of translating solver output into playable content Finally, system integration with an industry standard editor and an interface for editing semantic data and supporting mixed-initiative work would enable in-editor evaluation by game design professionals.

In order to investigate and develop the approaches necessary for modular ASP-based content generation for games, a range of investigative projects have been undertaken.

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 ontology (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.

Initial study: Read data from ontology, generate basic 2d level from assemblable components. Proof of concept using OWLCpp to read from ontologies and generate with C++ stubs. Set up DLVHex for further development of the concept; begun work using existing plugins to read from ontology and developing custom plugins to augment generation. Investigation into using ASP for complex layout solving.
Future plans: further develop custom plugins, and annotate ontology with ASP fragments. Define transformation process for iterative refinement. Provide a base ontology defining abstract base classes and common relations. Unreal plugin with editor interfaces. Proposal: plugin for Unreal, an industry standard editor. Interface to allow high-level definition of ontology data, where classes derive from provided 'base' ontology, annotated with ASP fragments. Implement a range of small ASP modules and DLVHex plugins for a selection of common tasks such as Gaussian random numbers or graph generation. Define a general transformation scheme guided by ontology structure to iteratively define the model of the level.

The initial proof-of-concept was a basic roguelike dungeon layout generator based upon `dlvhex`, an ASP solver which supports plugins for external computation. Basic semantic knowledge about dungeon constraints was represented via an external ontology and collision-checking

functions that could be queried by `dlvhex` during solving, and output was rendered in ASCII.

The use of ASP allowed certainty that properly constructed constraints regarding feasible paths from start to end would hold valid in all generated instances, whilst also potentially supporting more complex requirements such as a desired branching factor or the presence of loops and short-cuts within the game level.

Preliminary investigations have been 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. OWL provides an ontology inheritance mechanism, which allows for the system to include a base ontology to be extended with the domain specific concepts and parameters. This base ontology could 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.

An initial project was conducted using *owlcpp*, an open-source library developed in C++ to allow efficient and scalable parsing and reasoning with OWL 2 ontologies[3](Levin and Cowell, 2015). The generation of a simple rogue-like dungeon environment (as in Smith and Bryson, 2014) was chosen as an initial project with sufficient scope for simple generation tasks based on interesting hierarchical relations between concepts, and formal constraints such as spatial relations, connectivity and the existence of a valid start and end point. Relevant concepts and constraints were modelled in an OWL ontology and retrieved using *owlcpp*, with simple C++ stub functions used to generate appropriate values for unspecified data, and a basic output renderer for a visual ASCII representation of the generated content.

The sample output shown in Fig. 7-3 demonstrates some of the strengths and weaknesses of the prototype system. Room sizes and locations are within acceptable parameters, and rooms are connected to appropriate neighbours in a sensible fashion, without overlapping and with a valid route available from start (`@`) to end (`%`). However, the stub C++ corridor generation approach used for linking rooms that are specified as 'Connected' is overly simplistic, and does not respect the constraint that all corridors must be spatially disjoint. For the purposes of integration with a game engine, it was necessary to have a version of the library built with 64-bit address support, however this proved infeasible, and an alternative system was selected with support for both ASP and OWL input and reasoning: `dlvhex`.

*owlcpp* uses a Boost-based build system to produce libraries that may then be included in a C++ project, however a small number of the project's dependencies were not available with 64-bit support, hindering UE4 integration. `dlvhex` is primarily developed on Linux systems and

[3] `http://owl-cpp.sourceforge.net/` — accessed 27 November 2020

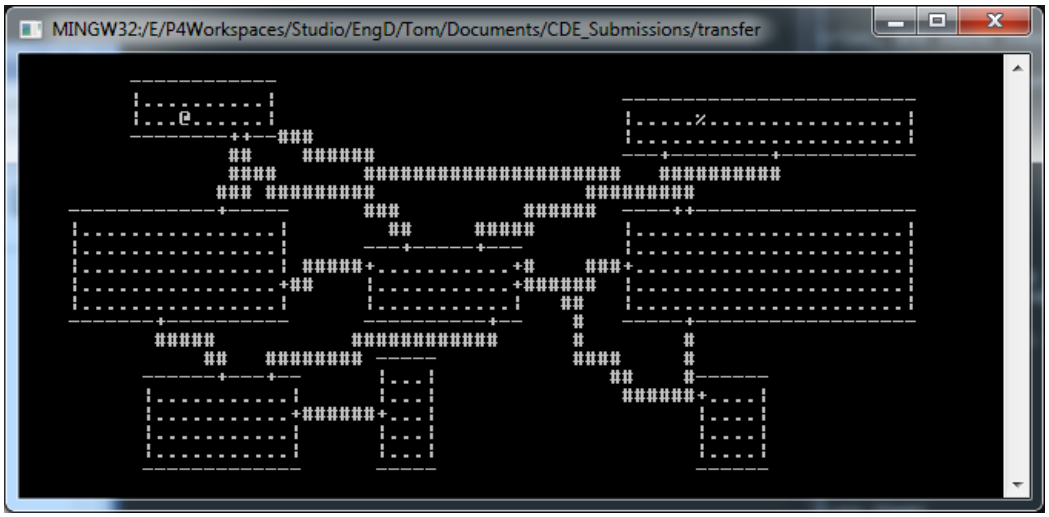



Figure 7-3: Sample output from *owlcpp* concept project with ontology data + stub-based generation.

is therefore provided with a makefile-based build system to produce the monolithic executable and the various plugin libraries, however legacy project files are also included for building previous versions on Windows. A range of alterations were required in order to compile and execute `dlvhex` and the `dlplugin` on Windows using Visual Studio (Ninja Theory's preferred development environment) prior to the proof-of-concept work described in Sec. 7.2.3, and further work was undertaken in order to convert the system into a 64-bit library with the correct compilation options for integration with UE4.

Though ASP provides useful guarantees regarding gameplay requirements, it is inefficient for some specific generation tasks such as solving geometric constraints. The `dlvhex` system supports plugins to provide external computation such as collision checking or path planning, and can also access external sources of data such as ontologies or other databases[4] (Eiter et al., 2006). `dlvhex` uses an extended ASP syntax known as HEX-programs, whereby 'external atoms' defined by plugins may have their truth values determined by calls to external sources such as databases, sensors or other non-ASP code, and then be reasoned over using Clingo (see Chap. 3). A number of existing plugins are available, including `dlplugin` which defines atoms for retrieving and reasoning over data specified in OWL. Further custom plugins may be defined in C++ or Python in order to provide support for functionality that ASP is not natively well-suited for — in the Rogue-like dungeon example this could include selection of numerical values according to a specified distribution, or efficient intersection detection.

Fig. 7-4 presents a room layout generated by ASP-style HEX code using the same ontology as in Fig. 7-3. Though it is perhaps less visually impressive than the *owlcpp* output due to the lack of an appropriate mechanism for laying out a visible representation of the connections between rooms, it demonstrates that a range of the ASP constraints and HEX plugin atoms are working

[4] `http://www.kr.tuwien.ac.at/research/systems/dlvhex/` — accessed 27 November 2020

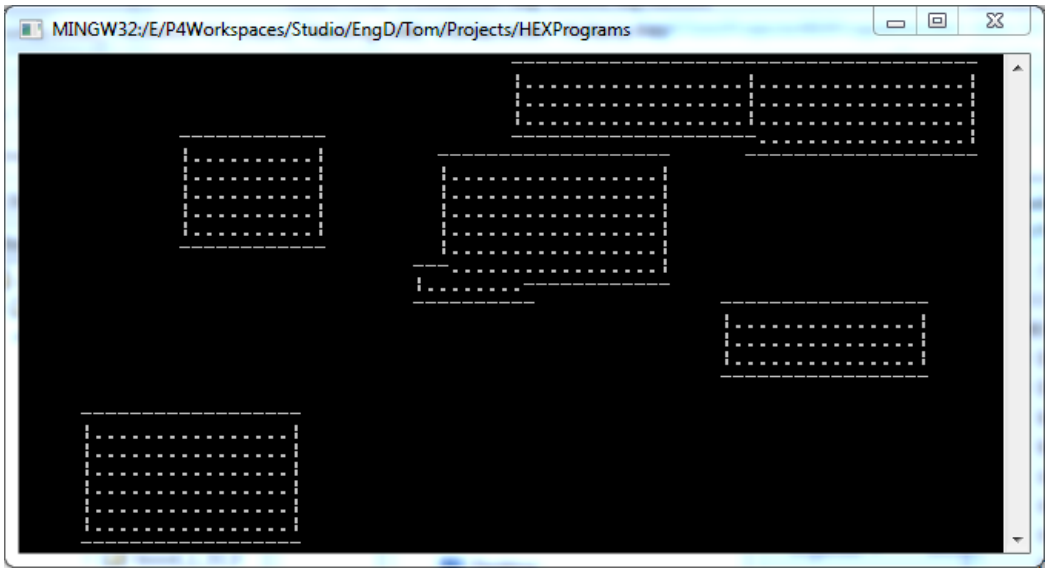



Figure 7-4: Sample output from `dlvhex` concept project with ontology data + ASP-based generation.

correctly, if not exactly as intended: the *internal* floor-space of individual rooms never overlap, and room sizes and locations are appropriately generated via plugin calls to suitable Gaussian or uniform distributions respectively. However there are a range of aesthetic considerations noticeably violated due to under-specification of suitable constraints – room walls overlap as they are not included in the spatial-disjointness constraint (which only considers floor-space to be part of the room), and rooms are not always well distributed through the available space. It is possible to model these concerns in ASP, as well as constraints ensuring validity of paths from start to end points, weighted preferences about branching factors, the existence of loops or dead-ends, and also objectives such as valid placement of keys, locks and rewards.

Both description logics (ontologies) and logic programming (ASP) provide reasoning-amenable formalisations of facts relating to a particular area, however they do so in different ways. One of the research challenges of this project will be developing ways to integrate the two approaches.

Though `dlvhex` supports querying OWL ontologies, for simplicity during development the current implementation specifies domain-specific information (about enemy types, difficulties, etc.) as ASP facts within the problem encoding. Though this is feasible for small-scale problems, it is an inflexible approach that does not facilitate decoupling between the generator and specification, and does not scale well for larger domains.Following the discussion in 7.2.3, it would be beneficial to store domain information in an OWL ontology, and retrieve it for use in ASP where necessary. As a concrete example, the some games provides two distinct player characters, A and B, with access to differing weapons and combat styles. There are also a small number of enemies specifically developed to be appropriate for B's abilities. If information about these differences and additional enemies were stored in the ABoxes of separate ontologies with a shared game TBox, then changing the behaviour of the generator to produce appropriate

waves for each character would be as simple as loading the correct ontology.

## 7.3 Conclusion

The research presented in this thesis lies at the intersection of academia and an industry company's demonstrated needs. A partial primary benefit of research in this context is the accumulation and dissemination into industry of knowledge of academic techniques, approaches and learnings; a secondary benefit is closer insight into industry practices and available needs and niches. In the preceding chapters we have presented a survey of relevant literature in the field, presented and explained our core technology, illustrated its application to an initial problem relevant to our industry partners, developed this approach to tackle a larger problem with direct comparison to prior academic work, and explored application of a standard PCG evaluation approach. Finally, we have summarised the contributions and a range of promising directions for further work.

In this thesis we have described the integration of ASP with a commercial engine and the generation of content in the form of combat wave progressions. This algorithmic approach potentially offers benefits over the traditional trial and evaluation techniques, as it allows designers to think in terms of a population of possible valid wave progressions, and quickly re-generate new progressions in response to design changes. Further, we discussed ways in which this work may be extended in order to make better use of the possibility afforded by ASP to generate complex constraint-driven outputs, and potentially provide a more responsive experience for the player.

We have also developed an approach for procedurally generating action-adventure game level greyboxes using transformation via ASP constraint solving. We present the refinement of a simple structured model of a level (the local and non-local events on path to completion) into a more detailed abstract model of the level (a directed graph showing areas within the level by challenge type, and key and lock events), and discuss further refinement including level elaboration for choice and optional areas, area refinement from known constraints, and integration with a game engine. We apply a quantitative analysis in order to investigate the expressive range of the initial formulation, and compare this system to a previous grammar-based generation approach for similar content. We note that using ASP enables us to easily carve out desired areas of the generative space whilst also continuing to satisfy hard gameplay- or implementation-related constraints.

# References


Abela, R., Liapis, A. and Yannakakis, G.N., 2015. A constructive approach for the generation of underwater environments. *Proceedings of the fdg workshop on procedural content generation in games*. Citeseer.

Abuzuraiq, A.M., Ferguson, A. and Pasquier, P., 2019. Taksim: A constrained graph partitioning framework for procedural content generation. *2019 ieee conference on games (cog)*. IEEE, pp.1–8.

Alvarez, A., Dahlskog, S., Font, J., Holmberg, J., Nolasco, C. and Österman, A., 2018. Fostering creativity in the mixed-initiative evolutionary dungeon designer [Online]. *Proceedings of the 13th international conference on the foundations of digital games*. New York, NY, USA: ACM, FDG '18, pp.50:1–50:8. Available from: `https://doi.org/10.1145/3235765.3235815`.

Antoniou, G. and Van Harmelen, F., 2004. Web ontology language: OWL. *Handbook on ontologies*. Springer, pp.67–92.

Bak, J., Nowak, M. and Jedrzejek, C., 2013. Graph-based editor for SWRL rule bases. *Ruleml (2)*. citeseer, pp.1–12.

Baldwin, A., Dahlskog, S., Font, J.M. and Holmberg, J., 2017. Towards pattern-based mixed-initiative dungeon generation [Online]. *Proceedings of the 12th international conference on the foundations of digital games*. New York, NY, USA: ACM, FDG '17, pp.74:1–74:10. Available from: `https://doi.org/10.1145/3102071.3110572`.

Bicho, F. and Martinho, C., 2018. Multi-dimensional player skill progression modelling for procedural content generation [Online]. *Proceedings of the 13th international conference on the foundations of digital games*. New York, NY, USA: ACM, FDG '18, pp.1:1–1:10. Available from: `https://doi.org/10.1145/3235765.3235774`.

Boenn, G., Brain, M., De Vos, M. and Ffitch, J., 2011. Automatic music composition using answer set programming. *Theory and practice of logic programming*, 11(2-3), pp.397–427.

Boulton, A., Hourizi, R., Jefferies, D. and Guy, A., 2017. A little bit of frustration can go a long way. *Advances in computer games - 15th international conferences, acg 2017, revised selected papers*. Springer, Springer Verlag, Lecture Notes in Computer Science (including subseries

Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pp.188–200. Available from: https://doi.org/10.1007/978-3-319-71649-7_16.

Brain, M., Cliffe, O. and De Vos, M., 2009. A pragmatic programmer's guide to answer set programming. *Software engineering for answer set programming (sea09)*, pp.49–63.

Brain, M. and De Vos, M., 2008. Answer set programming — a domain in need of explanation. *Exact08: International workshop on explanation-aware computing*.

Brewka, G., Eiter, T. and Truszczyński, M., 2011. Answer Set Programming at a glance. *Communications of the acm*, 54(12), pp.92–103.

Brown, M., 2017. *How my boss key dungeon graphs work* [Online]. Accessed: 2018-06-29. Available from: https://www.patreon.com/posts/how-my-boss-key-13801754.

Butler, E., Andersen, E., Smith, A.M., Gulwani, S. and Popović, Z., 2015. Automatic game progression design through analysis of solution features. *Proceedings of the 33rd annual acm conference on human factors in computing systems*. ACM, pp.2407–2416.

Butler, E., Smith, A.M., Liu, Y.E. and Popovic, Z., 2013. A mixed-initiative tool for designing level progressions in games. *Proceedings of the 26th annual acm symposium on user interface software and technology*. ACM, pp.377–386. Available from: https://doi.org/10.1145/2501988.2502011.

Calimeri, F., Fink, M., Germano, S., Ianni, G., Redl, C. and Wimmer, A., 2013. AngryHEX: an artificial player for Angry Birds based on declarative knowledge bases. *Popularize artificial intelligence*, p.29.

Canossa, A. and Smith, G., 2015. Towards a procedural evaluation technique: Metrics for level design. *Proceedings of the 2015 conference on the foundations of digital games(fdg 2015). monterey, ca, june.*

Card, O.S., 1985. *Ender's game*. 97th ed. Atom.

Carmichael, G. and Mould, D., 2014. A framework for coherent emergent stories. *Foundations of digital games*. p.6.

Caylı, M., Karatop, A.G., Kavlak, A.E., Kaynar, H., Türe, F. and Erdem, E., 2007. Solving challenging grid puzzles with answer set programming. *Proceedings of the 4th international workshop on answer set programming (asp'07)*. Universidade do Porto, Faculdade de Ciencias.

Čertický, M., 2013. Implementing a wall-in building placement in StarCraft with declarative programming. *arxiv preprint arxiv:1306.4460*, pp.1–17.

Cheng, W., 2017. Procedural enemy waves. *Procedural generation in game design*. AK Peters/CRC Press, chap. 14, pp.143–152.

Clarke, A.C., 1956. *The city and the stars*. London: Gollancz-Orion.

Cliffe, O., 2007. *Specifying and analysing institutions in multi-agent systems using answer set programming*. Ph.D. thesis. University of Bath.

Compton, K., 2016. *The hardest and most central problem in AI is, and may always be, this: Expressing, to a machine, the world we want.* [Twitter] [Online]. 23 December. Available from: `https://twitter.com/GalaxyKate/status/812512193891729408` [Accessed 13 September 2019].

Compton, K., Filstrup, B. and Mateas, M., 2014. Tracery: Approachable story grammar authoring for casual users. *Seventh intelligent narrative technologies workshop*.

Compton, K. and Mateas, M., 2015. Casual Creators. *Proceedings of the sixth international conference on computational creativity*, p.228.

Compton, K. and Mateas, M., 2017. A generative framework of generativity. *Thirteenth artificial intelligence and interactive digital entertainment conference*.

Compton, K., Osborn, J.C. and Mateas, M., 2013. Generative methods. *The fourth procedural content generation in games workshop, pcg*.

Compton, K., Smith, A. and Mateas, M., 2012. Anza island: Novel gameplay using ASP. *Proceedings of the 3rd workshop on procedural content generation in games*. ACM, p.13.

Cook, M., Eladhari, M., Nealen, A., Treanor, M., Boxerman, E., Jaffe, A., Sottosanti, P. and Swink, S., 2016. Pcg-based game design patterns. *Corr* [Online], abs/1610.03138. `1610.03138`, Available from: `http://arxiv.org/abs/1610.03138`.

Cook, M., Gow, J. and Colton, S., 2016. Danesh: Helping bridge the gap between procedural generators and their output. *Proceedings of the 7th international workshop on procedural content generation in games*.

Dahlskog, S., Björk, S. and Togelius, J., 2015. Patterns, dungeons and generators. *Foundations of digital games conference, fdg*. Foundations of Digital Games.

Dahlskog, S., Togelius, J. and Nelson, M.J., 2014. Linear levels through n-grams. *Proceedings of the 18th international academic mindtrek*, pp.200–206.

Dart, I.M., De Rossi, G. and Togelius, J., 2011. SpeedRock : procedural rocks through grammars and evolution. *Proceedings of the 2nd international workshop on procedural content generation in games*, pp.7–10. Available from: `https://doi.org/10.1145/2000919.2000927`.

Deterding, S., 2013. Skill atoms as design lenses for user-centered gameful design. *Workshop papers chi2013*.

Dormans, J., 2010. Adventures in level design: Generating missions and spaces for action adventure games. *Proceedings of the 2010 workshop on procedural content generation in games*. ACM, New York, NY, USA: ACM, PCGames '10, pp.1:1–1:8.

Dormans, J., 2011. Level design as model transformation: a strategy for automated content generation [Online]. *Proceedings of the 2nd international workshop on procedural content generation in games*. ACM, p.2. Available from: http://dl.acm.org/citation.cfm?id=2000921.

Dormans, J., 2017. Cyclic generation. *Procedural generation in game design*. AK Peters/CRC Press, chap. 9, pp.83–95.

Eiter, T., Ianni, G., Schindlauer, R. and Tompits, H., 2006. dlvhex: A prover for semantic-web reasoning under the answer-set semantics. *Ieee/wic/acm international conference on web intelligence*. IEEE, pp.1073–1074.

Epic Games, 2014. *Unreal Engine 4* (v.4.16.3) [computer program]. Available from: https://www.unrealengine.com/what-is-unreal-engine-4 [Accessed 1 October 2019].

Foreman-Mackey, D., 2016. corner.py: Scatterplot matrices in python. *The journal of open source software* [Online], 24. Available from: https://doi.org/10.21105/joss.00024.

Gaggl, S.A., Schweizer, L. and Rudolph, S., 2015. Bound your models! How to make OWL an ASP modeling language. *Proceedings of IULP 2015*.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T., 2014. Clingo=ASP + control: Preliminary report. *Corr* [Online], abs/1405.3694. Available from: http://arxiv.org/abs/1405.3694.

Green, M.C., Khalifa, A., Alsoughayer, A., Surana, D., Liapis, A. and Togelius, J., 2019. Two-step constructive approaches for dungeon generation [Online]. *Proceedings of the 14th international conference on the foundations of digital games*. New York, NY, USA: Association for Computing Machinery, FDG '19. Available from: https://doi.org/10.1145/3337722.3341847.

Green, M.C., Khalifa, A., Barros, G.A.B., Nealen, A. and Togelius, J., 2018. Generating levels that teach mechanics [Online]. *Proceedings of the 13th international conference on the foundations of digital games*. New York, NY, USA: ACM, FDG '18, pp.55:1–55:8. Available from: https://doi.org/10.1145/3235765.3235820.

Grey, D., 2017. When and why to use procedural generation. *Procedural generation in game design*. AK Peters/CRC Press, chap. 1, pp.3–12.

Gygax, G., Arneson, D. and Mentzer, F., 1983. *Dungeons & dragons: Players manual*. TSR Hobbies.

Hastings, E.J., Guha, R.K. and Stanley, K.O., 2009. Evolving content in the Galactic Arms Race video game. *Computational intelligence and games, 2009. cig 2009. ieee symposium on*. IEEE, pp.241–248.

Heijne, N. and Bakkes, S., 2017. Procedural zelda: A pcg environment for player experience research [Online]. *Proceedings of the 12th international conference on the foundations of digital games*. New York, NY, USA: ACM, FDG '17, pp.11:1–11:10. Available from: `https://doi.org/10.1145/3102071.3102091`.

Hendrikx, M., Meijer, S., Van Der Velden, J. and Iosup, A., 2013. Procedural content generation for games: A survey. *Acm transactions on multimedia computing, communications, and applications (tomm)*, 9(1), p.1.

Horn, B., Dahlskog, S., Shaker, N., Smith, G. and Togelius, J., 2014. A comparative evaluation of procedural level generators in the Mario AI framework. *Proceedings of the 9th international conference on foundations of digital games*. Society for the Advancement of the Science of Digital Games.

Horswill, I.D. and Foged, L., 2012. Fast procedural level population with playability constraints. *Aiide*.

Interactive Data Visualization, Inc., 2002. *SpeedTree for Games* (v.9) [computer program]. Available from: `https://store.speedtree.com/` [Accessed 1 October 2019].

Jacobsen, E.J., Greve, R. and Togelius, J., 2014. Monte Mario: platforming with MCTS. *Proceedings of the 2014 annual conference on genetic and evolutionary computation*. ACM, pp.293–300.

Karavolos, D., Bouwer, A. and Bidarra, R., 2015. Mixed-initiative design of game levels: Integrating mission and space into level generation. *Proceedings of the 10th international conference on the foundations of digital games*.

Kartal, B., Sohre, N. and Guy, S.J., 2016. Data driven sokoban puzzle generation with monte carlo tree search [Online]. *Twelfth artificial intelligence and interactive digital entertainment conference*. Available from: `http://motion.cs.umn.edu/r/sokoban-pcg`.

Karth, I. and Smith, A.M., 2017. Wavefunctioncollapse is constraint solving in the wild [Online]. *Proceedings of the 12th international conference on the foundations of digital games*. New York, NY, USA: ACM, FDG '17, pp.68:1–68:10. Available from: `https://doi.org/10.1145/3102071.3110566`.

Kerssemakers, M., Tuxen, J., Togelius, J. and Yannakakis, G.N., 2012. A procedural procedural level generator generator. *2012 ieee conference on computational intelligence and games (cig)*. IEEE, pp.335–341.

Knublauch, H., Musen, M.A. and Rector, A.L., 2004. Editing description logic ontologies with the protégé owl plugin. *International workshop on description logics*. vol. 49.

Koster, R., 2005. *A theory of fun for game designers*. Paraglyph Press, Scottsdale, Arizona.

Lavender, B. and Thompson, T., 2015. The Zelda dungeon generator: Adopting generative grammars to create levels for action-adventure games. *Todo*.

Lavender, B. and Thompson, T., 2016. A generative grammar approach for action-adventure map generation in The Legend of Zelda. *Proceedings of 7th international symposium for ai & games*. AISB.

Lavender, R., 2016. *The Zelda Dungeon Generator: Adopting generative grammars to create levels for action-adventure games*. Bachelor's thesis. University of Derby.

Levin, M.K. and Cowell, L.G., 2015. *owlcpp*: a C++ library for working with OWL ontologies. *Journal of biomedical semantics*, 6(1), pp.1–9.

Liapis, A., Martínez, H.P., Togelius, J. and Yannakakis, G.N., 2013. Adaptive game level creation through rank-based interactive evolution [Online]. *2013 ieee conference on computational inteligence in games (cig)*. IEEE, pp.1–8. Available from: `http://julian.togelius.com/Liapis2013Adaptive.pdf`.

Liapis, A., Smith, G. and Shaker, N., 2016. Mixed-initiative content creation. In: N. Shaker, J. Togelius and M.J. Nelson, eds. *Procedural content generation in games: A textbook and an overview of current research*. Springer, chap. 11, pp.195–214.

Liapis, A., Yannakakis, G.N. and Togelius, J., 2013a. Sentient sketchbook: Computer-aided game level authoring. *Proceedings of the 8th international conference on the foundations of digital games (fdg 2013)* [Online], pp.213–220. Available from: `http://www.itu.dk/people/anli/mixedinitiative/sentient_sketchbook.pdf`.

Liapis, A., Yannakakis, G.N. and Togelius, J., 2013b. Towards a generic method of evaluating game levels. *Aiide*.

Linden, R. van der, Lopes, R. and Bidarra, R., 2013. Designing procedurally generated levels. *Ninth artificial intelligence and interactive digital entertainment conference*.

Linden, R. van der, Lopes, R. and Bidarra, R., 2014. Procedural Generation of Dungeons. *Ieee transactions on computational intelligence and ai in games*, 6(1), pp.78–89. Available from: `https://doi.org/10.1109/TCIAIG.2013.2290371`.

Ludomotion, 22 February 2017. *Unexplored* [computer program]. Available from: `https://www.ludomotion.com` [Accessed 1 October 2019].

Marino, J.R., Reis, W.M. and Lelis, L.H., 2015. An empirical evaluation of evaluation metrics of procedurally generated Mario levels. *Todo*.

Mawhorter, P. and Mateas, M., 2010. Procedural level generation using occupancy-regulated extension. *Computational intelligence and games (cig), 2010 ieee symposium on*. IEEE, pp.351–358.

Mazeika, S., 2015. *ASP: It's basically creating a multitude of possible universes, and then CULLING THOSE THAT ARE NOT TO YOUR LIKING.* [Twitter] [Online]. 7 November. Available from: `https://twitter.com/StellaMazeika/status/663247947123986433` [Accessed 13 September 2019].

Nelson, M.J. and Smith, A.M., 2016. ASP with applications to mazes and levels. In: N. Shaker, J. Togelius and M.J. Nelson, eds. *Procedural content generation in games: A textbook and an overview of current research.* Springer, chap. 8.

Neufeld, X., Mostaghim, S. and Perez-Liebana, D., 2015. Procedural level generation with answer set programming for general video game playing. *2015 7th computer science and electronic engineering conference (ceec).* IEEE, pp.207–212.

Ninja Theory Ltd., 2013. *DmC: Devil May Cry* [computer program]. Available from: `https://ninjatheory.com/?page_id=12` [Accessed 1 October 2019].

Ninja Theory Ltd., 2015. *DmC: Devil May Cry: Definitive Edition* [computer program]. Available from: `https://ninjatheory.com/?page_id=1315` [Accessed 1 October 2019].

Nintendo, 2001. *The Legend of Zelda: Oracle of Seasons* [computer program]. [Accessed 1 October 2019].

Nitsche, M., Ashmore, C., Hankinson, W., Fitzpatrick, R., Kelly, J. and Margenau, K., 2006. Designing procedural game spaces: A case study. *Proceedings of futureplay.*

Perez-Liebana, D., Lucas, S.M., Gaina, R.D., Togelius, J., Khalifa, A. and Liu, J., 2019. *General video game artificial intelligence.* Morgan & Claypool Publishers. `https://gaigresearch.github.io/gvgaibook/`.

Preuss, M., Liapis, A. and Togelius, J., 2014. Searching for good and diverse game levels. *Computational intelligence and games (cig), 2014 ieee conference on.* IEEE, pp.1–8.

Reis, W.M., Lelis, L.H. and Gal, Y., 2015. Human computation for procedural content generation in platform games. *Conference of comp. intell. and games. ieee.*

Risi, S., Lehman, J., D'Ambrosio, D.B., Hall, R. and Stanley, K.O., 2015. Petalz: Search-based procedural content generation for the casual gamer. *Ieee transactions on computational intelligence and ai in games*, 8(3), pp.244–255.

Roddenberry, G., 1987. Star trek: The next generation [television series].

Rozen, R. van and Heijn, Q., 2018. Measuring quality of grammars for procedural level generation [Online]. *Proceedings of the 13th international conference on the foundations of digital games.* New York, NY, USA: ACM, FDG '18, pp.56:1–56:8. Available from: `https://doi.org/10.1145/3235765.3235821`.

Schreier, J., 2019. *Zelda: Link's Awakening's Chamber Dungeons Are A Big Disappointment* [Online]. Available from: https://kotaku.com/zelda-link-s-awakenings-chamber-dungeons-are-a-big-dis-1838038985 [Accessed 2022-06-24].

Schwab, B., Mark, D., Dill, K., Lewis, M. and Evans, R., 2011. GDC: Turing tantrums: AI developers rant [Online]. Available from: https://www.gdcvault.com/play/1014586/Turing-Tantrums-AI-Developers-Rant [Accessed 29 November 2022].

Shaker, N., Liapis, A., Togelius, J., Lopes, R. and Bidarra, R., 2016. Constructive generation methods for dungeons and levels. In: N. Shaker, J. Togelius and M.J. Nelson, eds. *Procedural content generation in games: A textbook and an overview of current research.* Springer, chap. 3, pp.31–55.

Shaker, N., Nicolau, M., Yannakakis, G.N., Togelius, J. and Neill, M.O., 2012. Evolving levels for Super Mario Bros using grammatical evolution. *Computational intelligence and games (cig), 2012 ieee conference on.* IEEE, pp.304–311.

Shaker, N., Smith, G. and Yannakakis, G.N., 2016. Evaluating content generators. In: N. Shaker, J. Togelius and M.J. Nelson, eds. *Procedural content generation in games: A textbook and an overview of current research.* Springer, chap. 12, pp.215–224.

Shaker, N., Togelius, J. and Nelson, M.J., 2016. *Procedural content generation in games: A textbook and an overview of current research.* Springer.

Shepard, J., 2017. Level design ii: Handcrafted integration. *Procedural generation in game design.* AK Peters/CRC Press, chap. 7, pp.63–71.

Short, E., 2016. Bowls of oatmeal and text generation. *Emily short's interactive storytelling* [Online]. Available from: https://emshort.blog/2016/09/21/bowls-of-oatmeal-and-text-generation/ [Accessed 12 December 2022].

Side Effects Software, 1996. *Houdini FX* (v.19) [computer program]. Available from: https://www.sidefx.com/products/houdini/ [Accessed 15 November 2022].

Smelik, R.M., Tutenel, T., Bidarra, R. and Benes, B., 2014. A survey on procedural modelling for virtual worlds. *Computer graphics forum.* Wiley Online Library, vol. 33, pp.31–50.

Smelik, R.M., Tutenel, T., Kraker, K.J. de and Bidarra, R., 2010. Declarative terrain modeling for military training games. *International journal of computer games technology*, 2010, p.2.

Smith, A.J. and Bryson, J.J., 2014. A logical approach to building dungeons: Answer set programming for hierarchical procedural content generation in roguelike games. *Proc. 50th anniversary convention of the aisb.*

Smith, A.M., 2012. *Mechanizing exploratory game design.* Thesis (PhD.). University Of California Santa Cruz.

Smith, A.M., Andersen, E., Mateas, M. and Popović, Z., 2012. A case study of expressively constrainable level design automation tools for a puzzle game [Online]. *Proceedings of the international conference on the foundations of digital games*. ACM, New York, NY, USA: ACM, FDG '12, pp.156–163. Available from: `https://doi.org/10.1145/2282338.2282370`.

Smith, A.M., Butler, E. and Popović, Z., 2013. Quantifying over play: Constraining undesirable solutions in puzzle design. *Proceedings of the international conference on the foundations of digital games*. pp.221–228.

Smith, A.M., Lewis, C., Hullett, K., Smith, G. and Sullivan, A., 2011. An inclusive taxonomy of player modeling. *University of california, santa cruz, tech. rep. ucsc-soe-11-13*.

Smith, A.M. and Mateas, M., 2011. Answer set programming for procedural content generation: A design space approach. *Computational intelligence and ai in games, ieee trans. on*, 3(3), pp.187–200.

Smith, A.M., Nelson, M.J. and Mateas, M., 2010. Ludocore: A logical game engine for modeling videogames. *Computational intelligence and games (cig), 2010 ieee symposium on*. IEEE, pp.91–98.

Smith, G., 2014. Understanding procedural content generation: a design-centric analysis of the role of pcg in games. *Proceedings of the 32nd annual acm conference on human factors in computing systems*. ACM, pp.917–926. Available from: `https://doi.org/10.1145/2556288.2557341`.

Smith, G., 2017. Understanding the generated. *Procedural generation in game design*. AK Peters/CRC Press, chap. 22, pp.231–243.

Smith, G. and Whitehead, J., 2010. Analyzing the expressive range of a level generator. *Proceedings of the 2010 workshop on procedural content generation in games*. ACM, p.4.

Smith, G., Whitehead, J. and Mateas, M., 2010. Tanagra: A mixed-initiative level design tool. *Proceedings of the fifth international conference on the foundations of digital games*. ACM, pp.209–216.

Smith, G., Whitehead, J. and Mateas, M., 2011. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *Computational intelligence and ai in games, ieee trans. on*, 3(3), pp.201–215.

Smith, T., Padget, J. and Vidler, A., 2016. Design space descriptions for logical generation of content. *Stairs 2016 — proceedings of the eighth european starting ai researcher symposium*. IOS Press, pp.209–214.

Smith, T., Padget, J. and Vidler, A., 2018. Graph-based generation of action-adventure dungeon levels using answer set programming. *Proceedings of the 13th international conference on the foundations of digital games*. ACM, p.52. Available from: `https://doi.org/10.1145/3235765.3235817`.

Summerville, A., 2018. Expanding expressive range: Evaluation methodologies for procedural content generation. *Fourteenth artificial intelligence and interactive digital entertainment conference*.

Togelius, J., Champandard, A.J., Lanzi, P.L., Mateas, M., Paiva, A., Preuss, M. and Stanley, K.O., 2013a. Procedural Content Generation: Goals, Challenges and Actionable Steps. *Artificial and computational intelligence in games*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, *Dagstuhl Follow-Ups*, vol. 6, pp.61–75.

Togelius, J., Justinussen, T. and Hartzen, A., 2012. Compositional procedural content generation [Online]. *Proceedings of the 3rd workshop on procedural content generation in games*. ACM, p.16. Available from: `http://julian.togelius.com/Togelius2012Compositional.pdf`.

Togelius, J., Karakovskiy, S., Koutník, J. and Schmidhuber, J., 2009. Super Mario evolution. *Computational intelligence and games, 2009. cig 2009. ieee symposium on*. IEEE, pp.156–161.

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

Togelius, J. and Shaker, N., 2016. The search-based approach. In: N. Shaker, J. Togelius and M.J. Nelson, eds. *Procedural content generation in games: A textbook and an overview of current research*. Springer, chap. 2, pp.17–30.

Togelius, J., Shaker, N. and Dormans, J., 2016. Grammars and l-systems with applications to vegetation and levels. In: N. Shaker, J. Togelius and M.J. Nelson, eds. *Procedural content generation in games: A textbook and an overview of current research*. Springer, chap. 5, pp.73–98.

Togelius, J., Shaker, N., Karakovskiy, S. and Yannakakis, G.N., 2013b. The mario ai championship 2009-2012. *Ai magazine*, 34(3), pp.89–92.

Togelius, J., Yannakakis, G.N., Stanley, K.O. and Browne, C., 2011b. Search-based procedural content generation: A taxonomy and survey. *Computational intelligence and ai in games, ieee transactions on*, 3(3), pp.172–186.

Trescak, T., Esteva, M. and Rodriguez, I., 2010. A virtual world grammar for automatic generation of virtual worlds. *The visual computer*, 26(6-8), pp.521–531.

Tutenel, T., Bidarra, R., Smelik, R.M. and De Kraker, K.J., 2009a. Rule-based layout solving and its application to procedural interior generation. *Proceedings of the CASA Workshop on 3D advanced media in gaming and simulation*.

Tutenel, T., Bidarra, R., Smelik, R.M. and Kraker, K.J.D., 2008. The role of semantics in games and simulations. *Computers in entertainment (cie)*, 6(4), p.57.

Tutenel, T., Linden, R. van der, Kraus, M., Bollen, B. and Bidarra, R., 2011. Procedural filters for customization of virtual worlds. *Proceedings of the 2nd international workshop on procedural content generation in games*. New York, NY, USA: ACM, PCGames '11, pp.5:1–5:8.

Tutenel, T., Smelik, R., Bidarra, R. and Kraker, K.J. de, 2009b. Using semantics to improve the design of game worlds. *Proceedings of the AAAI conference on artificial intelligence and interactive digital entertainment*. AIIDE, vol. 5, pp.100–105.

Valtchanov, V. and Brown, J.A., 2012. Evolving dungeon crawler levels with relative placement. *Proceedings of the fifth international c* conference on computer science and software engineering*. ACM, pp.27–35.

Valve Software, 18 April 2011. *Portal 2* [computer program]. Available from: `http://www.thinkwithportals.com/` [Accessed 1 October 2019].

Whitehead, J., 2020. Spatial layout of procedural dungeons using linear constraints and SMT solvers. *International conference on the foundations of digital games*. pp.1–9.

Yannakakis, G.N. and Liapis, A., 2016. Searching for surprise. *Proceedings of the international conference on computational creativity*.

Yannakakis, G.N., Liapis, A. and Alexopoulos, C., 2014. Mixed-initiative co-creativity. *Proceedings of the 9th conference on the foundations of digital games*.

Yu, D., 2016. *Spelunky: Boss fight books# 11*, vol. 11. Boss Fight Books.

Zagal, J.P. and Bruckman, A., 2008. The game ontology project: Supporting learning while contributing authentically to game studies. *Proceedings of the 8th international conference on international conference for the learning sciences-volume 2*. pp.499–506. Available from: `http://www.fisme.science.uu.nl/en/icls2008/283/paper283.pdf`.

Zagal, J.P., Mateas, M., Fernández-Vara, C., Hochhalter, B. and Lichti, N., 2007. Towards an ontological language for game analysis. *Worlds in play: International perspectives on digital games research*, 21, p.21.

# Appendices

These documents were produced as part of the preparation to run a quantitative user study, as detailed in Sec. 7.2.1. Note: these documents are presented here unaltered save for heading and page numbers — this means they include the lead researcher's previous name, which has changed since they were generated.

## A. Data plan

A completed data plan is one of the requirements for approval by the Ethics Committee of any user study at the University of Bath. The document is autogenerated following a questionnaire within the DMPonline software.

## B. Participant consent form

The production of a participant consent form that concisely conveys the participant's rights including the right of withdrawal etc., is also part of the ethical approval process.

## C. Participant info sheet

The Ethics Board approval process requires that we produce a thorough explanation of the study to be provided to each participant to take away, as part of the process of informed consent.

# Procedural Content Generation for Computer Games - User Study

*A Data Management Plan created using DMPonline*

Creators: Thomas Smith, masjap@bath.ac.uk

Affiliation: University of Bath

Template: Engineering and Physical Sciences Research Council (EPSRC)

ORCID iD: 0000-0001-9032-652X

Project abstract:

A software tool has been produced to aid in the development of prototype 3D environments for games, subject to gameplay/designerly constraints. Data will be collected as part of a user study investigating usage of the tool for a small number of typical scenarios: after a briefing and preliminary questionnaire to elicit relevant experience(s), users will be recorded whilst undertaking a brief tutorial and a small number of tasks using the tool. Participants will be encouraged to speak aloud about their intentions and impressions, and on completion of the interactive portion of the study will be debriefed and given an opportunity to share further thoughts or ask any remaining questions. Information relating to the session including video, audio, software logs and screencaptures will be recorded, and transcribed after completion of the study in preparation for analysis and conclusion-drawing.

Last modified: 04-10-2018

# Procedural Content Generation for Computer Games - User Study

## Data Collection

What data will you collect or create?

Video, audio and screencapture recording of user study session. Precise formats to be determined by available hardware and software; probably .mp4.

Handwritten notes on user study session observations, scanned to digital format as .png or .jpg.

Software log of user study session, output as .csv.

Transcription of user study session recordings as plain text (.txt).

Total volume of data expected to be 1-3Tb; proportions likely to be primarily recordings (~90%), images of scanned notes (~6%), logs and plain text transcripts (~4%).

How will the data be collected or created?

Initial recordings will be collected via webcam, microphone and screen recording software during the user study, and saved to individual participant folders on portable storage.

Software logs will be output to the test machine during the user study and then moved to individual participant folders on the portable storage.

Investigator observations will be handwritten during the user study, and then scanned and saved to appropriate participant folders on the portable storage.

Transcription of recordings and observation notes will be performed after the user study, with common actions and insights from the software logs reduced to a controlled vocabulary.

## Documentation and Metadata

What documentation and metadata will accompany the data?

Due to the nature of the study and the personal nature of individual participant recordings, data collected in this study is unlikely to be viable for reuse in other research contexts. Interview templates, a copy of the participant brief and a copy of this data management plan may be stored with the dataset in order to provide context, but production of structured specialist metadata will not be necessary.

## Ethics and Legal Compliance

How will you manage any ethical issues?

Participants will be briefed on the intended uses of the data, and on protocols associated with its storage and preservation. Participation will be voluntary, and informed consent requested before any data collection occurs. Identifiable recordings will be stored in anonymised folders on encrypted portable storage with a secure password, and kept no longer than two

years past the completion of the study.

How will you manage copyright and Intellectual Property Rights (IPR) issues?

Data ownership and intellectual property rights will be assigned in accordance with the existing agreement between Ninja Theory ltd, the Centre for Digital Entertainment, and the Principal Investigator. Informtation collected will contain neither proprietary nor patentable data, and will be neither shared nor licensed for reuse, due to the personally identifiable nature of the recordings and the inapliccablility of the data for purposes beyond the current study.

## Storage and Backup

How will the data be stored and backed up during the research?

The data will be primarily stored on an encrypted portable drive with secure password known only to the Principal Investigator, and accessed only from devices with secure passwords and regular virus/malware scans.
To guard against the possibility of accidental data loss, a backup of the data collected during the study (but not the transcripts of the recordings) will be stored on an encrypted disk image with secure password known only to the Principal Investigator, and left in the possession of the industry partner company. A second encrypted backup may also be made to the university's X: drive storage, if appropriate.
Once produced, a copy of the recording transcripts will be encrypted with a secure shared password and shared with the co-investigators.

Limited non-identifiable portions of the transcripts, agreggate conclusions reached from the data, and insights relating to elements of the study may be published and released as part of a thesis or paper(s) relating to the study, and these may be stored insecurely on collaborative editing platforms, however no sensitive data will be reproduced in this fashion.

How will you manage access and security?

Access to the data will be restricted to the Principal Investigator only, through the use of encryption and secure passwords. Personal data will not be transferred or backed up via the cloud or third-party services.
An encrypted backup of the user study recordings will be left in a geographically distinct secure location with the partner company, but will not be accessible to them.
Encrypted copies of the anonymised transcripts will be shared with the co-investigators.

## Selection and Preservation

Which data are of long-term value and should be retained, shared, and/or preserved?

None of the primary data collected (user study recordings) are likely to be of long-term value, or will require preservation/archival beyond the lifespan of the research project plus a maximum of a two-year period of prudential retainment. Due to the necessarily personally identifiable nature of the recordings, excluding them from long-term retention is justified, and this simplifies the consent process for participants.

Non-personally-identifiable anonymised transcripts of the recordings should be retained for substantiation of the research findings, for a minimum of 10 years.

What is the long-term preservation plan for the dataset?

Anonymised copies of the user study transcripts will be preserved in the University's Research Data Archive, and offered to the UK Data Service (https://www.ukdataservice.ac.uk/), in accordance with the relevant policies.

## Data Sharing

How will you share the data?

Anonymised copies of the user study transcripts will be preserved in the University's Research Data Archive, and offered to the UK Data Service (https://www.ukdataservice.ac.uk/), in accordance with the relevant policies.

Are any restrictions on data sharing required?

There are no foreseen restrictions on data sharing required. Data will be made available through the relevant repositories within 12 months of the end of the project, subject to consultation with the partner company confirming that the data has no commercially confidential value.

## Responsibilities and Resources

Who will be responsible for data management?

The postgraduate research supervisor (Julian Padget) will be the Data Steward for the project, responsible for data capture, metadata production, storage & backup management, and secure sharing of appropriate portions of the data. All of these responsibilities and activities will be delegated to the postgraduate research student (Thomas Smith), with the exception of the continuing obligation to secure any shared portions of the data. The external collaborating partner (Andrew Vidler) will share in the obligation to secure any shared portions of the data.

What resources will you require to deliver your plan?

The postgraduate research student's research fund will cover the purchase of one (1) portable storage drive through approved purchasing mechanisms, for secure storage and transportation of the collected data. The partner company will provide for storage of an encrypted backup of the recorded data for the remaining duration of the project at no financial cost, however in the event that this is not possible the research fund will cover the purchase of one (1) secondary drive for this purpose. Deposit/repository charges are not anticipated.


# CONSENT FORM

## Procedurally generating content for game development using Answer Set Programming

Thomas Smith (t.a.e.smith@bath.ac.uk) Supervisor: Dr Julian Padget (j.a.padget@bath.ac.uk)

**Please initial box if you agree with the statement**

1. I have been provided with information explaining what participation in this project involves. ☐
2. I have had an opportunity to ask questions and discuss this project and have received satisfactory answers to all questions I have asked. ☐
3. I have received enough information about the project to make a decision about my participation. ☐
4. I understand that I am free to withdraw my consent to participate in the project at any time without having to give a reason for withdrawing. ☐
5. I understand that I am free to withdraw my data within two weeks of my participation. ☐
6. I understand the nature and purpose of the procedures involved in this project. These have been communicated to me on the information sheet accompanying this form. ☐
7. I understand the data I provide will be treated as confidential, and that identifying information will not be disclosed in any presentation or publication of the research. ☐
8. I understand that my consent to use the data I provide is conditional upon the University complying with its duties and obligations under the Data Protection Act. ☐
9. I hereby fully and freely consent to my participation in this project. ☐

Participant's signature: _____ Date: _____

Participant name in BLOCK Letters: _____

Researcher's signature: _____ Date: _____

Researcher name in BLOCK Letters: _____

If you have any concerns or complaints related to your participation in this project please direct them to the Department Research Ethics Officer for Computer Science, Prof. Steve Payne (s.j.payne@bath.ac.uk).


# PARTICIPANT INFORMATION SHEET

## Procedurally generating content for game development using Answer Set Programming

Name of Researcher: Thomas Smith
Contact details of Researcher: t.a.e.smith@bath.ac.uk

Name of Supervisor: Dr Julian Padget
Contact details of Supervisor: j.a.padget@bath.ac.uk

This information sheet forms part of the process of informed consent. It should give you the basic idea of what the research is about and what your participation will involve. Please read this information sheet carefully and ask one of the researchers named above if you are not clear about any details of the project.

### 1. What is the purpose of the project:

We are conducting a study that explores the effectiveness of a prototype software system developed to generate greybox level prototypes according to designer-imposed constraints.

### 2. Why have I been selected to take part?

You have been invited because you have working experience with the use of Unreal Engine 4 to develop and modify level designs.

### 3. Do I have to take part?

No. It is completely up to you to decide if you would like to participate. You can ask questions about the research before making your decision. If you do agree to participate, we will describe the project and go through this information sheet with you. If you agree to take part, we will then ask you to sign a consent form. However, you may withdraw yourself from the study at any time, without giving a reason, by advising the researchers of this decision.

**4. What will I have to do?**

You will be invited to select a convenient time for participating in the study, where you will be briefed on the study and software and given an opportunity to ask any questions you may have. If you are still happy to take part, you will be asked to sign a consent form indicating that you understand what will be involved and are happy to be recorded.

There will be a brief tutorial and questionnaire to assess relevant experience with similar software. You will then be given a small number of tasks to complete using the software, and an opportunity for self-driven exploration of its capabilities. You will be encouraged to speak aloud about your intentions and impressions, and on completion of the interactive portion of the study you will be debriefed and given an opportunity to share further thoughts or ask any remaining questions. Information relating to the session including video, audio, software logs and screen-captures will be recorded, and transcribed after completion of the study. The duration of the session is expected to be 45-60 minutes.

**5. What are the exclusion criteria?**

There are no indications that participation in the study is not recommended for any defined class of possible participants. However, if you feel unsafe or uncomfortable at any time you are free to withdraw from the study by informing the researcher.

**6. What are the possible benefits of taking part?**

There are no direct benefits of taking part in the project. However, the information that you and other participants provide in this project will help us to assess the potential of this prototype software, and form part of the assessment of the tool.

**7. What are the possible disadvantages and risks of taking part?**

There are no disadvantages or expected risks to you taking part in the project. If the study involves a question or task that you do not want to answer or complete for any reason, you can choose not to do so.

**8. Will my participation involve any discomfort or embarrassment?**

We do not expect you to feel any discomfort or embarrassment if you take part in the project. If however you do feel uncomfortable or appear upset at any time, the researcher will stop the study straight away and may direct you to approach an appropriate support service.

**9. Who will have access to the information that I provide?**

Only the research team will have access to information that you provide. All records will be treated as confidential.

**10. What will happen to the data collected and results of the project?**

The information you provide as part of the study is the *research data*. Any research data from which you can be identified (i.e. your name and the audio/video recording of the interview), is known as *personal data*. It does not include data where the identity has been removed (anonymous data). We will minimise our use of personal data in the study as much as possible.

*Personal data* and *research data* will be stored confidentially on an encrypted, password-protected drive. Interview recordings will be anonymised and transcribed, and will not be kept for longer than 5 years. Members of the research team (Thomas Smith, Julian Padget, Andrew Vidler) will have access to research data. Anonymised transcripts may be transferred to, and stored at, a destination outside the European Economic Area. Any such data transfer will be done securely and with a similar level of data protection as required under UK law. Your name or other identifying information will not be disclosed in any presentation or publication of the research.

Your consent form will be stored for three years, and anonymised copies of the user study transcripts will be preserved in the University's Research Data Archive, and offered to the UK Data Service (https://www.ukdataservice.ac.uk/), in accordance with the relevant policies.

We would like your permission to make the anonymised data available for use in future studies, and to share data with other researchers (e.g. in online research data archives, as detailed above). All personal information that could identify you will be removed or changed before information is shared with other researchers or results are made public. We would also like your permission to use anonymous direct quotes in research publications.

**11. Who has reviewed the project?**

This project has been approved by the Head of Department for Computer Science, University of Bath (Eamonn O'Neill) [reference: EIRA1-2359].

**12. How can I withdraw from the project?**

If you wish to stop participating before completing all parts of the study, you can inform one of the above identified researchers by email, telephone or in person. You can withdraw from the project at any point without providing reasons for doing so and without consequence to yourself.

If, for any reason, you wish to withdraw your data after the study has been undertaken, please contact an identified researcher within two weeks of your participation. After this date, it may not be possible to fully withdraw your data due to anonymisation and aggregation. Your individual input however will not be identifiable in any way in any presentation or publication.

**13. What happens if there is a problem?**

If you have a concern about any aspect of this study, please speak to the researcher (t.a.e.smith@bath.ac.uk) or their supervisor (j.a.padget@bath.ac.uk), who will do their best to answer your query. The researcher should acknowledge your concern within 10 working days and give you an indication of how they intend to deal with it. If you remain unhappy or wish to make a formal complaint, please contact the Department Research Ethics Officer for Computer Science at the University of Bath who will seek to resolve the matter in a reasonably expeditious manner (Professor Steve Payne, s.j.payne@bath.ac.uk).

**14. If I require further information who should I contact and how?**

Thank you for expressing an interest in participating in this project. Please do not hesitate to get in touch with us if you would like some more information.

Name of Researcher: Thomas Smith
Contact details of Researcher: t.a.e.smith@bath.ac.uk

Name of Supervisor: Dr Julian Padget
Contact details of Supervisor: j.a.padget@bath.ac.uk