Nobody's A Critic: On The Evaluation Of Creative Code Generators – A Case Study In Videogame Design

Michael Cook, Simon Colton and Jeremy Gow
Computational Creativity Group, Imperial College, London

Abstract

Application domains for Computational Creativity projects range from musical composition to recipe design, but despite all of these systems having computational methods in common, we are aware of no projects to date that focus on program code as the created artefact. We present the Mechanic Miner tool for inventing new concepts for videogame interaction which works by inspecting, modifying and executing code. We describe the system in detail and report on an evaluation based on a large survey of people playing games using content it produced. We use this to raise issues regarding the assessment of code as a created artefact and to discuss future directions for Computational Creativity research.

Introduction

Automatic code generation is not an unusual concept in computer science. For instance, many types of machine learning work because of an ability to generate specialised programs in response to sets of data, e.g., logic programs (Muggleton and de Raedt 1994). Also, evolutionary systems can be seen to produce code either explicitly, in the case of genetic programming, or implicitly through evolutionary art software that uses programmatic representations to store and evaluate populations of artworks. Moreover, in automated theory formation approaches, systems such as HR (Colton 2002) generate logic programs to calculate mathematical concepts. These programs are purely for representation, however, rather than in pursuit of creative programming. In software engineering circles, 'metaprogramming' is used to increase developer efficiency by expanding abstract design patterns, or to increase adaptability by reformatting code to suit certain environments. None of these instances of code generation fully embrace the act of programming for what it is – a creative undertaking. There can be no field better placed to appreciate programming in this way than Computational Creativity.

Building software that can generate new software, or modify its own programming, opens up huge new areas for Computational Creativity, as well as enriching all existing lines of research by allowing us to reflect on our systems as potential artefacts of code generators or modifiers themselves. We attempt here to highlight some of these future opportunities and challenges by describing the design of a prototype system, *Mechanic Miner* (Cook et al 2013), which designs a particular videogame element – game mechanics – by inspecting, modifying and executing Java game code.

Mechanic Miner produced game mechanics for *A Puzzling Present*, a platform game released in December 2012 and downloaded more than 5900 times. This game included survey and logging code to assess, among other things, the quality of the mechanics generated by Mechanic Miner in terms of perceived enjoyability and the challenge in using them. In analysing the data and evaluating the system, however, we have noticed issues with current notions of assessment within Computational Creativity research, and how they interact with the idea of evaluating a creative system whose output is program code. We explore these issues below. In this paper we make the following contributions:

- We describe the development of a creative system that generates code as its output.
- We report on the first large-scale experimental evaluation of interactive computationally-created artefacts.
- We discuss issues involving the assessment of creative systems working in media with a high barrier to entry.

The rest of this paper is organised as follows: in *Mechanic Miner – Overview* we describe Mechanic Miner in full, detailing how it generates and evaluates new game mechanics through code. In *A Puzzling Present – Evaluation Through Play* we describe *A Puzzling Present*, a game designed and released using mechanics invented by Mechanic Miner. We discuss the difficulties in evaluating interactive code, how a balance can be struck between presenting a survey and offering a natural experience to the user, and present some results from our survey. In the section *Creativity in Code Generation*, we highlight issues for the future of code generation, as well as promising opportunities for Computational Creativity. in *Related Work* we briefly describe previous approaches to mechanic generation and highlight why code generation is necessary to advance in this area. Finally, in *Conclusions* we review our achievements and reflect on where our work with game mechanics will lead next.

Mechanic Miner – Overview

Definitions

Many conflicting definitions exist for game mechanics, as described, for instance, in (Sicart 2008), (Kelly 2010) and (Cook 2006). For our purposes here, we define a game mechanic as a piece of code that is executed whenever a button is pressed by the player that causes a change in the game's state. How a game mechanic is defined in code will vary

from game to game, depending on the architecture of the game engine, the way the game has been coded within that engine, and the idiosyncrasies of the individuals who wrote the rest of the game code. For example, below is a line of code from a game written in the *Flixel* game engine. Executing the code causes the player character to jump, by adding a fixed value to its velocity (the player's gravity will counteract this change over time and bring the character to the ground again).

```
player.velocity.y -= 400;
```

Mechanic Miner generates artefacts within a subspace of game mechanics, which we have called *Toggleable Game Mechanics* (TGMs). A TGM is an action the player can take to change the state of a variable. That is, given a variable v and a modification function f with inverse f^{-1}, a TGM is an action the player can take which applies $f(v)$ when pressed the first time, and $f^{-1}(v)$ when pressed a second time. The action may not be perfectly reversible; if v is changed elsewhere in the code between the player taking actions f and f^{-1}, the inverse may not set v back to the value it had when f was applied to it. For instance, if v is the player's x co-ordinate, and the player moves around after applying f, then their x co-ordinate will not return to its original value after applying f^{-1}, as it was modified by the player moving.

Generation Mechanic Miner is written in Java, and therefore able to take advantage of the language's built-in Reflection features that allow program code to inspect and explore other code[1]. For example, the following code retrieves a list of fields of a given class:

```
MyClass.getClass().getFields()
```

Such Field objects can be manipulated to yield their name, their type, or even passed to objects of the appropriate type to find the value of that field within the object. Java has similar objects to represent most other language features, such as `Method`s and generic types. Given the definition of a TGM above, we can see that Reflection allows software to store the location of a target field at runtime, and dynamically alter its value. Using the Reflections library, Mechanic Miner can therefore obtain a list of all classes currently loaded, and iterate through them asking for their available fields. It can use information on the type of each field to conditionally select modifiers that can be applied to the field.

Java's Reflection features do not provide encapsulation for primitive operations such as mathematical operators, assignment or object equality. To solve this problem, we created custom classes to represent these operations, which enabled Mechanic Miner to select modifiers for a field that could be applied during evaluation. Thus, a TGM is composed of a `java.lang.Field` object, and a type-specific `Modifier`. For example, a mechanic that doubled the x co-ordinate of the player object would use the `org.flixel.FlxSprite` object's `x` field, and an `IntegerMultiplyModifier` defined as follows:

[1]We further extended this core functionality by employing the Reflections library from http://code.google.com/p/reflections.

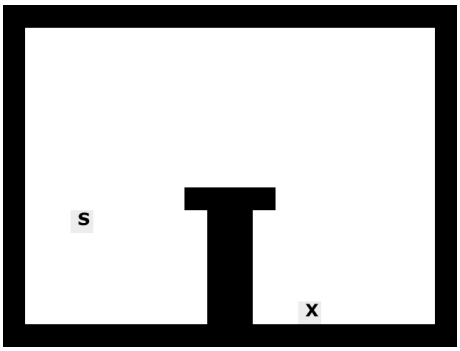

Figure 1: A sample level used to evaluate mechanics.

```
public void apply(Field f){
 if(toggled_on){
  f.setValue(f.getValue()*coefficient);
 } else{
  f.setValue(f.getValue()/coefficient);
 }
}
```

Where `coefficient` is set to 2 in the case of doubling, but can be set by Mechanic Miner to an arbitrary value as it evaluates potential mechanics. Note the use of a boolean flag, `toggled_on`, to retain the state of the TGM so that its effect can be reversed. Modifiers were selected to give a coverage of key operations that might be performed on fields, such as inverting the value of a boolean field, adding or multiplying values for a numerical field, or setting numerical fields to exact values (such as zeroing a field, and then returning it to its original value). Future extensions we plan to the generation process will allow for the use of mathematical discovery tools such as HR (Colton 2002) that could invent calculations which transform the values of the fields.

Evaluation In order to evaluate generated mechanics, we need strong criteria that describe the properties that desirable mechanics should have. In the version of Mechanic Miner described here, we focus purely on the *utility* of a mechanic (that is, whether it affords the player new possibilities when playing the game) rather than how fun the mechanic is to use, how easy it is to understand, or how appropriate it is for the context. Utility is not only easy to define, but can be defined in absolute terms, which provides a solid target for a system to evaluate towards.

To illustrate how utility can be identified by Mechanic Miner, consider the game level shown in Figure 1. The player starts in the location marked 'S' and must reach the location marked 'X', and when they do, we say that the player has *solved* the game level. The game operates similar to a simple game such as *Super Mario*; the player is subject to gravity, but can move left and right as well as jumping a small distance up. Under these rules alone, the level is not solvable because the central wall is too high and impedes progress. Therefore, if we were to add a new game

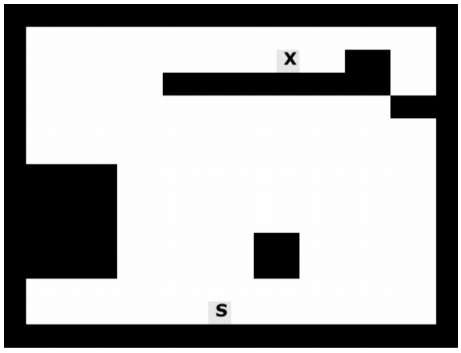

Figure 2: A level generated by Mechanic Miner for the 'gravity inversion' mechanic. The player starts in the 'S' position and must reach the exit, marked 'X'.

mechanic such as the inversion of gravity, and as a result the level were to become solvable, we could conclude that the new mechanic had expanded the player's abilities, and allowed them to solve a level of this type.

This idea is central to Mechanic Miner's evaluation of mechanics – it uses a solver to play game levels in a breadth-first fashion, trying legal combinations of button presses while remaining agnostic to what mechanics the buttons relate to. It will continue to search for combinations of button presses until it finds at least one solution; at this point it continues looking for combinations of that length, completing the breadth first expansion of this depth, and will then return a list of all paths that led to a solution. Hence it can try arbitrary mechanics without knowing in advance what the associated code does when executed. This enables it to firmly conclude whether the mechanic has contributed to the player's abilities by assessing which areas of the level are accessible that were not previously, which in turn enables it assess the level itself.

Level Generation Mechanic Miner's ability to simulate gameplay in order to evaluate mechanics can also be applied in reverse to act as a fitness function when generating levels for specific mechanics using evolutionary techniques. Representing a level design as a 20x15 array of blocks that are either solid or empty, we can evaluate the fitness of a level with respect to a mechanic M by playing the level twice – once with only the basic controls available, and once with M added to the controls. If the level is solvable with M, but *not* solvable without it, then the level is given a higher fitness. Using a binary utility function as our primary evaluation criteria strengthens the system's ability to provide exact solutions to the problem – either the level is completed, or it is not. In order to have a gradient between the two so that the evolutionary level designer can progress towards good levels, we moderate the fitness based on what proportion of the level was accessible. Thus, over time, levels that are more accessible emerge until eventually the exit is reachable from the start position (and thus the level is solvable).

Figure 2 shows a level generated for use with a mechanic called *gravity inversion*. Activating the mechanic would cause gravity to pull the player towards the ceiling instead of the floor. Activating it again would reverse the effect. Note that the level is not solvable without this mechanic, as the platforms are too high to jump onto.

The simulation-driven approach to level design allowed for the resulting software to be highly parameterised. Information such as the minimum number of distinct actions required to solve a level (where each button press is considered a distinct action) or the number of times a mechanic must be used, allowed the system to generate levels with different properties. It also allows the system to remain blind to the mechanic it is designing for. This allows Mechanic Miner to exploit created mechanics without having a human intervene and describe aspects of the mechanic to it, giving it greater creative independence as it is theoretically able to discover a wholly new mechanic in a H-creative way, and generate levels for that mechanic without any assistance. We can view this within the creativity tripod framework of (Colton 2008), which advocates implementing skill, appreciation and imagination in software. In particular, we see the ability to use output from one system to inspire creative work in another without external assistance as an example of skill as well as an appreciation of what makes a game mechanic useful to the player. We also claim that simulating player behaviour is in some sense *imagining* how they would play.

Illustrative Results

Below are examples of mechanics generated by Mechanic Miner. All of the effects can be reversed by the player:

- An ability to increase the player's jump height, allowing them to leap over taller obstacles.
- An ability to rubberise the player, making them able to bounce off platforms and ceilings.
- An ability to turn gravity upside down, sucking the player upwards.

These mechanics are evident in commercially successful games, such as Cavanagh's *VVVVVV* which featured gravity inversion as a core mechanic. Bouncing was an unexpected result for us, as we had no idea it was in the space of possibilities, although it has been featured in some games developed in other engines, particularly Nygren's *NightSky*. Cavanagh has received multiple nominations in the Independent Games Festival (IGF), and NightSky was shortlisted for Excellence In Design and the Grand Prize in the 2009 IGF.

Novel game mechanics are highly prized in game design circles. Many international design awards have tracks for innovative gameplay or mechanics (such as the IGF Nuovo Award[2]) and game design events often centre around the creation of unique methods of interaction (such as the Experimental Gameplay Workshop[3]). Mechanic Miner's ability to reinvent existing but niche mechanics is encouraging, given the small design space the system currently has access to.

As well creating mechanics, Mechanic Miner was also able to find exploits in the supplied game code, and use

[2]http://www.igf.com/

[3]http://www.experimental-gameplay.org/

them to create emergent gameplay – something which we had not anticipated as a capability of the system. One mechanic, which teleported the player a fixed distance left or right, was used by Mechanic Miner to design levels which at first glance had no legal solution. After inspecting the solution traces produced by the simulator, it became clear that the mechanic was being used in an innovative way to take advantage of a weakness in the code that described the player's jump. Jumping checked if the player's feet were in contact with a solid surface. By teleporting inside a wall, this check would be passed, and the player could jump upwards. Repeated applications of this technique allowed the player to jump up the side of walls – complicated exploitation of code, more commonly seen in high-end gameplay by *speedrunners*[4], i.e., gamers who compete over finding exploits in popular videogames to help them complete the games in the shortest time possible. For example, speed runs of the popular puzzle game *Portal* involve the abuse of 3D level geometry to escape the level's boundaries and pass through solid walls.

A Puzzling Present - Evaluation Through Play

To evaluate some of the mechanics and levels designed by the Mechanic Miner system, we developed a short compilation game featuring hand-selected mechanics, titled *A Puzzling Present* (APP). APP was released in late December 2012 on the Google Play store and desktop platforms[5]. The objective was to conduct a large-scale survey of players in order to gain feedback on the types of mechanic generated by the system, in addition to evaluating different metrics for level design. However, we were also conscious that interruptions to play, or overt presentation of the software as an experiment rather than a game, may deter players from completing levels or giving feedback and/or change the nature of the experiment, which is to ask their opinion on games, not surveys. In designing APP, we therefore made several trade-offs to balance these two factors.

All play sessions were logged in terms of which buttons the player presses, at what times, which can be used to fully replay a given player's attempt at a level. In addition to this, upon starting the game for the first time, the player was asked to opt-in to short surveys after each level. These took the form of two multiple-choice rating tasks on a 1-4 scale, evaluating enjoyability and difficulty. Figure 3 shows the survey screen. This presented itself to the player upon reaching the exit to a level, assuming the player had agreed to respond to surveys, although even in this case, they could continue without responding to the survey.

75614 sessions were recorded in total, over 5933 unique devices. When asked to opt-in to surveys, 60.7% of users agreed. Those who opted-in contributed 63.4% of the total session count. 92.3% of sessions played by opt-ins resulted in at least one of the two questions being answered, with 89.9% of sessions resulting in both questions being answered. Although the survey questions provided a rich source of data, by allowing us to gain qualitative evaluations

[4]Such as the community at http://speeddemosarchive.com/

[5]Download from www.gamesbyangelina.org/downloads/app.html.

Figure 3: Survey screen from *A Puzzling Present*

of the levels and game mechanics, the log data (which is recorded for all players) is equally valuable, and so allowing players who did not wish to participate in the survey to continue to play the game (or those who initially agreed to change their minds later) we gained an additional 32,000 level traces which we otherwise might have lost.

APP contained thirty levels, split into sets of ten that share a common mechanic. The three game mechanics are those described in the *Illustrative Results* section above: higher jump, bouncing and gravity inversion. Each level required the game mechanic to be used to complete it, but were generated using differing metrics for difficulty expressed through evolutionary parameters within the level designer. These were broken down as follows: two levels used a baseline setting determined through experimentation ('Baseline'); two levels put stricter requirements on minimum reaction times needed ('Faster Reaction'); two levels selected for longer paths from start to exit ('Longer Path'); two levels selected for more mechanic use in the shortest solutions ('Higher Mechanic Use'); and two levels selected for longer action chains in the solution. This provided a variety of the levels for the player to test, and allowed us to analyse feedback data to assess these metrics for future use. In order to mitigate bias or fatigue introduced as a result of experiencing certain levels or sets of levels before others, the order in which a particular player experienced the levels was randomised when the game was first started up. This was done by first randomising the order of the game mechanics, and then randomising the order of the ten levels within that set, thereby ensuring that all levels which share a mechanic are experienced together, to provide a more cohesive experience.

Figure 4 shows the mean difficulty and fun ratings for the nth level played as the people progressed through the 30 levels. These mean ratings remained fairly consistent throughout the game, with the exception of the 30th level. As levels were presented randomly, we assume this is an effect of the very low number of people still playing at this point. This consistency indicates that learning or fatigue did not seem to have much effect on player experience. This may be down to the interactivity of the artefact in question, and raises the question of whether the evaluation of created artefacts is more consistent when the survey participants are interactively engaged. We discuss this later as future work.

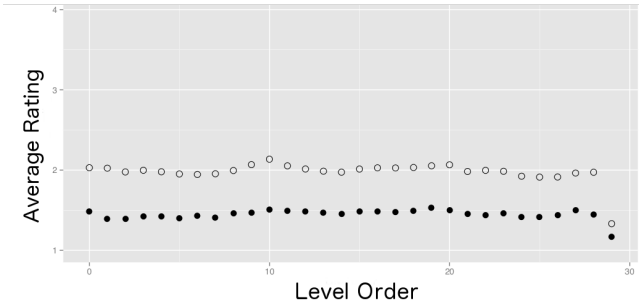

Figure 4: Mean fun (white circles) and difficulty (black circles) ratings for the nth level played.

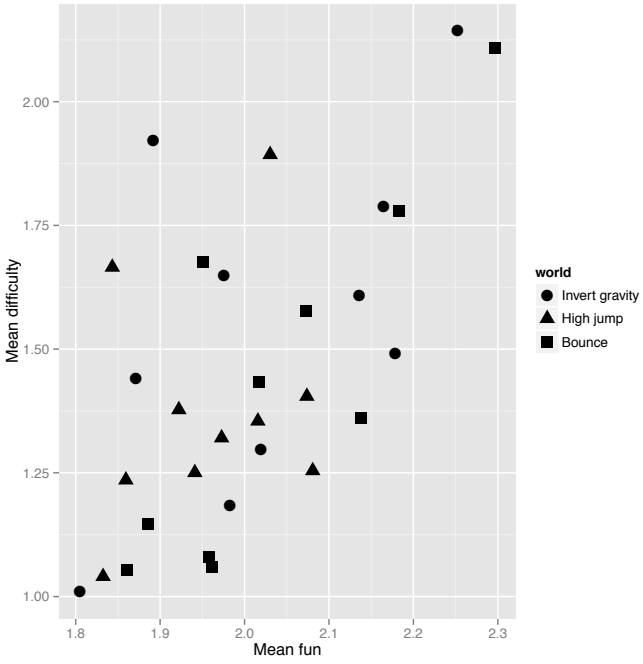

Figure 5: Mean level fun and difficulty, broken down by 'world' (a group of levels that share a mechanic).

The number of players completing a given set *(world)* of ten levels for a certain mechanic is consistent across the three game mechanics; 2259 completed World one, 2151 completed World two and 2219 completed World three. The data show no bias towards players not completing any particular one of the three worlds, suggesting that players left due to general fatigue with the system as a whole, rather than the content generated by Mechanic Miner. This may be down to the human-designed elements of the game that were common throughout the three worlds – such as the interface, control scheme, or artwork – and therefore not attributable to the output of Mechanic Miner.

Under statistical analysis of the survey scores, we found a moderate and highly significant rank correlation between mean difficulty and enjoyability (Spearman's $\rho = 0.56$, $p = 0.002$). The relationship between the difficulty of a level and the perceived enjoyability of a level is an interesting one to consider. While we might expect an inverse relationship for an audience who are easily frustrated with games, we also see many examples of games in which challenge correlates to an enjoyable game. We postulate that the correlation between mean difficulty and enjoyability exists here because the levels are, on average, *too* easy – the average difficulty rating across all levels is just 1.45, on a scale of 1 to 4 – and so an increase in difficulty was welcomed as it made the levels more interesting. A later study, with improved difficulty metrics to give a broader spread of skill levels, would help confirm this hypothesis.

Group	Mean Fun	Mean Difficulty
High Jump	1.96	1.38
Invert Gravity	2.02	1.55
Bounce	2.03	1.42
Baseline	1.96	1.30
Faster Reaction	2.01	1.51
Longer Path	1.95	1.20
Higher Mechanic Use	2.03	1.60
Longer Solution	2.06	1.66

Figure 6: Mean level fun and difficulty, broken down by game mechanic and level design parameters.

The mean fun and difficulty by world mechanic and level generation metric are shown in Table 6. Variations in mean fun are very small between groups, whereas mean difficulty shows greater separation, especially between the metrics. An analysis of variance (ANOVA) showed highly significant ($p < 0.001$) separate main effects for fun and difficulty with respect to both factors. There was also a significant interaction between mechanic and metric, which we do not report here. Post-hoc Tukey's HSD tests suggested the following significant differences between groups: a) the mechanics Invert Gravity and Bounce are more fun than High Jump; b) the metrics Fast Reaction, High Use and High Actions are more fun than Baseline and Longer Path; c) all differences in mean difficulty between mechanics, and between metrics, are significant.

Creativity In Code Generation

Nobody's a Critic

Many different approaches to assessing creativity in software have been proposed over the last decade of Computational Creativity research. Ritchie (2007) suggests that the creativity of a system might be established by considering what the system produces, evaluating the artefacts along such lines as *novelty*, *typicality* and *quality*. This leads to the proposal of ratios between sets of novel artefacts produced by a system, and sets that are of high quality, for instance. While this is helpful in establishing the performance of a given system, it presupposes both a minimum level of understanding in those assessing the system, and a direct connection between the means of interaction with the artefact, and the generated work itself.

In the case of software – particularly interactive media

whose primary purpose is entertainment – we are not guaranteed either of these. The consumers of software, such as those that evaluated *A Puzzling Present*, are often laypeople to the world of programming, even if they are highly experienced in interacting with software. More importantly, there is a disconnect between the presentation of code through its execution within a game environment, and the nature of the generated code itself. All software designed for use by the general public – from word processors to video games – presents a metaphorical environment in which graphics, audio and systems of rules come together to present a cohesive, interactive system with its own internal logic, symbols, language and fiction. In *A Puzzling Present* in particular, generated game mechanics operated on obscure variables hidden away within a complex class structure. To the interacting player, this is simply expressed as objects moving differently on-screen. This disconnect makes it hard for any user to properly evaluate the generated code itself, because they are not engaging with the underlying representation or mechanics of the software they are using.

Other approaches to evaluation consider the process of creation itself as crucial to the perception of creativity. In (Colton, Charnley, and Pease 2011) the authors propose the FACE model that considers elements of the creative process such as the generation of contextual information (which the authors call *framing*) and the use and invention of aesthetic judgements that affect creative decision-making. This focus on the process is a promising alternative to the artefact-heavy assessment methods that are more common in Computational Creativity, but problems abound here also, since in order to judge the creative process, a person must be able to comprehend that process to some degree.

As noted in (Johnson 2012), the majority of the systems in Computational Creativity have focused on 'old media' application domains, such as the visual arts, music and poetry. Although the skill ceiling for these media is undeniably high, they have very low barriers to entry. Most people have drawn pictures as children, attempted to crack new jokes, or hummed improvised ditties to themselves. While they may not exhibit even a small percentage of the virtuosity present at the top end of the medium in question, by engaging in the creation of artefacts, they can appreciate the process and are better placed to comment on it – or indeed they feel so, even if this is not the case. As a result, creative systems operating in the realm of old media often find truth in the term 'everyone's a critic'. By contrast, programming is a skill that is only recently being taught below university level in the western world; therefore, asking the general public to assess the creativity of a code generator by commenting on its creative process is unlikely to result in a useful or fair assessment.

This phenomenon – where nobody is a critic – makes it hard to apply existing thinking on the evaluation of creative systems to large-scale public surveys.

Speaking In Code

If neither the artefact-centric nor the process-centric approach is suitable to assess creative code generators, this begs the question of how we can proceed in assessing these systems on a large scale. We believe the key may be one particular element of the model described in FACE model of (Colton, Charnley, and Pease 2011), namely *framing* information that describes an artefact and the process that created it, as explored further in (Charnley, Pease, and Colton 2012).

Figure 7: Framing information in *Stealth Bastard*.

Code is not designed to be read by people. Extensive education is needed to understand the basics of programming structure and organisation, including additional time spent on learning specific languages. Even experienced programmers do not rely on these skills alone to understand program code – instead they leave plain-English comments so that others, and they themselves, will be able to understand the meaning of code long after it has been written. In interactive media, the need to explain features legibly and correctly situated within the (possibly fictional) context of the software is especially integral to the user's understanding and enjoyment of a piece of software. Video games, for instance, rely on their ability to create an immersive environment where all functionality is communicated through the fiction of the game world in question. The arcade game *Space Invaders* is not about co-ordinates overlapping and numbers being decremented – it is about shooting missiles at aliens and protecting your planet from attack.

This all amounts to a clear need to build into creative code generation systems the ability to explain the function of code it produces. This could be done either by annotating and describing the function of the raw code itself or, in the case of presenting artefacts to a layperson for assessment or consumption, by describing the function of the code in terms of the metaphors and context dictated by the software the code is part of. In the latter case, this poses interesting problems more akin to creative natural language generation. Videogames, for example, must describe the functionality of game mechanics in terms of what they enable the player to do within the game world - Figure 7 shows the *Stealth Bastard* game (Biddle 2012) explaining how to complete a level. Note the use of a physical verb (*enter*), a symbolic noun (*exit*) and a reference to meta-game objectives (*completing a level*). These are concepts unrelated to the technical specifics of game code, but crucial to the player's understanding of the thematic and ludic qualities of the game.

The generation of textual descriptions of both the creative process *and* the generated code is crucial in enabling these systems to be assessed by the general public. It will also become more important in autonomously creative systems that generate code for use in interactive contexts, where the meaning of the code must be conveyed clearly to a user. This

is a highly-prized feature of human-designed software[6] and is crucial in autonomously creative systems where artefacts are not subject to curation prior to their use.

Beyond Software

Considering program code as an artefact produced by a creative system allows us to reconsider existing creative systems as potential code generators themselves. Modules within creative systems might be able to integrate criteria such as those described in (Ritchie 2007) into a process of self-exploration and modification – where new code is created for generative submodules, and evaluated according to its ability to produce content along axes such as novelty, typicality or quality. Code generation should not be thought of, therefore, as a distinct strand of Computational Creativity that runs alongside other endeavours in art, poetry and the like. Instead, it should be viewed as a new lens through which to view existing takes on Computational Creativity, and a new way to improve the novelty and ingenuity of creative systems of all kinds.

If generic notions of novelty or typicality for code can be developed, then they can be applied across mediums to great effect. Comparisons of code segments have been explored within verification and software engineering approaches (Bonchi and Pous 2012; Turon et al. 2013), but for the purposes of Computational Creativity, a significantly different approach will be required, as we consider the ludic, aesthetic and semantic similarities in the output of a piece of code, rather than its raw data. If this can be done, creative software will no longer need to be considered static, instead empowered with the ability to generate new functionality within itself; creative artefacts will no longer need to be considered as finished when they leave a piece of software, but could improve and iterate upon their designs in response to use; and creative software will no longer be considered simply executing code written by humans, but instead be seen to be a collaborator in its own creation.

Related Work

The generation of game mechanics is closely related to the design of game rules in the more abstract sense. METAGAME (Pell 1992) is an early example of a system that attempted to generate new game rulesets. This worked by varying existing rulesets from well-known boardgames such as chess and checkers, using a simple grammar that could express the games as well as provide room for variation. Grammar-based approaches to ruleset generation are common in this area, perhaps most prominently seen in Ludi (Browne and Maire 2010) which evolved boardgame rulesets from a grammar of common operations, or work in (Togelius and Schmidhuber 2008) and (Cook and Colton 2012) which present similar work for realtime videogames.

Grammar-based approaches work well because they explore spaces of games that are defined by common core concepts; but are naturally limited by the nature of the human-designed grammar as a result. An alternative approach that can cover a broader space is to use annotated databases of mechanical components, and then assemble them to suit a particular design problem. Work in (Nelson and Mateas 2007) uses this approach to design games around simple noun-and-verb input, while (Treanor et al. 2012) use an annotated database approach to develop games that represent a human-defined network of concepts.

Smith and Mateas (2010) present an alternative approach, describing a generator of game rulesets without an evaluative component. The system they describe uses answer set programming to define a design space through a set of logical constraints. Solutions to these constraints describe game rulesets, therefore if constraints are chosen to restrict solutions to a certain space of good games, solving them will yield high-quality games. These criteria can be narrowed down by adding further constraints to the answer set program. This can be seen as somewhat related to grammatical approaches – higher-level concepts are defined by hand (such as 'character movement' or 'kill all') which are then selected for use later. This has similar limitations to the grammatical approaches, in that it is dependent on external input to define its initial language, and that this restricts the novelty of the system as a result. The future work proposed in (Smith and Mateas 2010) was to focus more on programmatic modification, however, which would have further distinguished the approach.

Conclusions and Further Work

We have described Mechanic Miner, a code modification system for generating executable content for videogames, and A Puzzling Present, a game which we released built using content generated by Mechanic Miner. We showed that code can be used as both a source material and a target domain for Computational Creativity research, and that it can lead to greater depth than working with metalevel abstractions of target creative domains, offering surprise and novelty even on a small scale. Through evaluation of gameplay responses, we drew conclusions about the presentation of creative artefacts to large audiences for evaluation. Finally, we raised the issue of how created artefacts can be evaluated by an audience which, in general, has no experience in the domain the artefacts reside within.

This work has also highlighted several areas of future work needed to expand the concepts behind Mechanic Miner to prove the worth of the approach in generating more sophisticated mechanics and games. These include work to expand the expressiveness of the code generation, so that it can include higher-level language concepts such as method invocation, expression sequences, control flow and object creation. This will lead to a large expansion of the design space, which will raise issues of efficiency and evaluation, also bearing further investigation.

We will also be using our experimental results to tune both our existing metrics for level and mechanic design, and to drive further development in systems such as Mechanic Miner, to increase their autonomy and their ability to seek out novel content. We are particularly interested in how different difficulty metrics can be combined to produce a diverse set of game content.

[6] E.g., as promoted in Apple's *Human Interface Guidelines*

We will also consider the looming problem of code generation's relationship with metaphorical gameplay. Game designer and critic Anna Anthropy describes games as "an experience created by rules" (Anthropy 2012). The way in which this experience is created, however, is deeply grounded in the player's ability to connect the systems inside a game with the real world. In *Super Mario*, for instance, eating a mushroom makes you larger, and conveys extra speed and jumping power. In the game's code, this is simply a collision of two objects, and some state changes. Notions such as size visually indicating strength or ability, or the idea that consuming food can improve your strength, are fundamentally connected to real-world knowledge, and less evident simply by looking at code. Discovering ways that software can discover these relationships for itself will be a major hurdle in developing code generators capable of designing meaningful game content, but also a gateway to an unprecedented level of creative power for software, and an opportunity to bring art, music, narrative and mechanics together in a more meaningful way than ever before.

The field of Computational Creativity was founded on the belief that computers could be used to simulate, enhance and investigate aspects of creativity, and researchers have created many complex pieces of software by hand. We believe that the time is ripe to move this a step further, and to turn the ideas we have developed on our own creations; to reconsider our artificial artists, composers and soup chefs as pieces of code that can be assessed, altered and improved at the same level of granularity that they were created. In order to do so, however, we may need to challenge some assumptions we hold about certain creative mediums and the relationship the general public has with them.

Acknowledgements

We would like to thank the reviewers for their input and suggestions, particularly regarding the discussion section. This research was funded by EPSRC grant EP/J004049.

References

Anthropy, A. 2012. *Rise of the Videogame Zinesters: How Freaks, Normals, Amateurs, Artists, Dreamers, Drop-outs, Queers, Housewives And People Like You Are Taking Back An Art Form*. Seven Stories Press.

Biddle, J. 2012. Stealth bastard deluxe.

Bonchi, F., and Pous, D. 2012. Checking NFA equivalence with bisimulations up to congruence. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13.

Browne, C., and Maire, F. 2010. Evolutionary Game Design. *IEEE Transactions on Computational Intelligence and AI in Games* 2(1):1–16.

Charnley, J.; Pease, A.; and Colton, S. 2012. On the notion of framing in Computational Creativity. In *Proceedings of the Third International Conference on Computational Creativity*.

Colton, S.; Charnley, J.; and Pease, A. 2011. Computational Creativity Theory: The FACE and IDEA models. In *Proceedings of the Second International Conference on Computational Creativity*.

Colton, S. 2002. *Automated Theory Formation in Pure Mathematics*. Springer.

Colton, S. 2008. Creativity versus the perception of creativity in computational systems. In *Proceedings of the AAAI Spring Symposiumo on Creative Intelligent Systems*.

Cook, M., and Colton, S. 2012. Initial results from co-operative co-evolution for automated platformer design. In *Proceedings of the Applications of Evolutionary Computation (EvoGames workshop)*.

Cook, M.; Colton, S; Raad, A; and Gow, J 2013. Mechanic Miner: Reflection-Driven Game Mechanic Discovery and Level Design. In *Proceedings of the Applications of Evolutionary Computation (EvoGames workshop)*.

Cook, D. 2006. What are game mechanics? http://www.lostgarden.com/2006/10/what-are-game-mechanics.html.

Johnson, C. 2012. The creative computer as romantic hero? or, what kind of creative personae do computational creativity systems exemplify? In *Proceedings of the Third International Conference on Computational Creativity*.

Kelly, T. 2010. Game dynamics vs game mechanics. http://www.whatgamesare.com/2010/12/game-dynamics-vs-game-mechanics.html.

Muggleton, S., and de Raedt, L. 1994. Inductive logic programming: Theory and methods. *Journal of Logic Programming* 19(20).

Nelson, M. J., and Mateas, M. 2007. Towards automated game design. In *Proceedings of the 10th Congress of the Italian Association for Artificial Intelligence*.

Pell, B. 1992. Metagame in symmetric, chess-like games. In *Heuristic Programming in Artificial Intelligence 3: The Third Computer Olympiad*.

Ritchie, G. 2007. Some empirical criteria for attributing creativity to a computer program. *Minds and Machines* 17(1):67–99.

Sicart, M. 2008. Defining game mechanics. *Game Studies*.

Smith, A., and Mateas, M. 2010. Variations forever: Flexibly generating rulesets from a sculptable design space of mini-games. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*, 273–280.

Togelius, J., and Schmidhuber, J. 2008. An experiment in automatic game design. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*.

Treanor, M.; Blackford, B.; Mateas, M.; and Bogost, I. 2012. Game-o-matic: Generating videogames that represent ideas. In *Proceedings of the Third Workshop on Procedural Content Generation in Games*.

Turon, A. J.; Thamsborg, J.; Ahmed, A.; Birkedal, L.; and Dreyer, D. 2013. Logical relations for fine-grained concurrency. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13.