From Mechanics to Meaning and Back Again: Exploring Techniques for the Contextualisation of Code

Michael Cook and Simon Colton
Computational Creativity Group
Goldsmiths College, University of London
http://www.doc.gold.ac.uk/ccg/

Abstract

Code generation is a promising new area for the automatic production of mechanics and systems in games. Generated code alone is not sufficient for inclusion in a rich, fully-designed game, however - it lacks context to bind the functionality of code to the metaphorical setting of the game. In this paper we explore potential solutions to this problem, both in terms of creative systems which co-operate with human content, and the possibility for contextual meaning in autonomous, human-free creative systems as well.

Introduction

In (Treanor 2013) the author describes the separation between the mechanics of a game and it's *instantial assets* - the visual and aural content that applies a theme to a set of mechanics. Through rebranding a simple arcade game Treanor shows how different sets of instantial assets can lead to different interpretations of a game, and also shows how different assignments of affect to game components can shift our perception of what is occurring when a particular cause and effect sequence is triggered in a game. Figure 1, from Treanor's work, shows four different versions of *Kaboom*, a simple arcade game. In the original version of the game, a bomber drops explosives from the top of the screen, which the player must catch before they reach the bottom of the screen. Failing to catch enough bombs ends the game. In Treanor's remappings, the visual theme shifts the story the game is telling, in some cases completely reversing the implications of the actions of the player.

In (Nelson and Mateas 2007) and (Nelson and Mateas 2008) the authors describe a system which can reskin a set of game mechanics dynamically, given nouns and verbs describing objects and their interactions with one another. This allows for the input phrase *shoot pheasant* to be linked to known verbs and nouns using synonym lookups, and ultimately transferred into potential game mechanics (such as a game where the player, controlling a duck, must avoid being shot at). Often, the system is capable of providing alternative designs for a game using the same input phrase. For example, for *shoot pheasant* the system can also produce a game

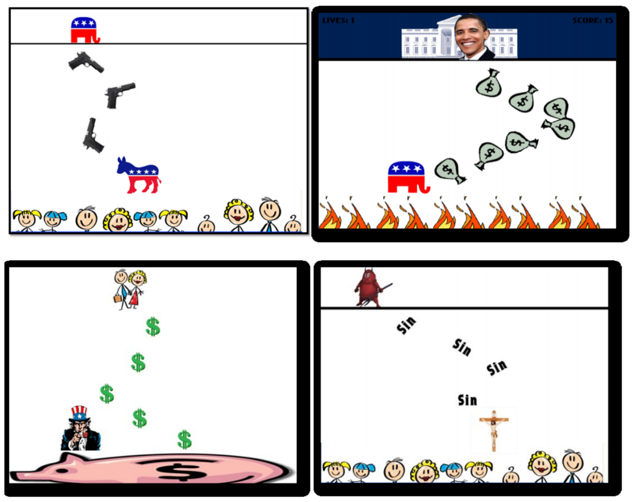

Figure 1: Four visual redesigns of the arcade game *Kaboom*, from (Treanor 2013).

where the player controls a crosshair and must shoot at passing ducks. Given enough initial information in its mechanics database, and a broad enough index of synonyms, the system can provide multiple recontextualisations of a game scenario - though the system is of course unable to invent new entries into its database, or make connections outside of its understanding of synonyms.

The notion of applying theming onto suitable game mechanics has been developed further by Treanor and others with the *Game-o-Matic* (Treanor et al. 2012a), a design tool which can take conceptual maps and select small game mechanics (which Treanor calls *micro-rhetorics* (Treanor et al. 2012b)) that can be composed together and tweaked to suit the concepts being expressed. As with Nelson and Mateas' work above, the system connects concepts provided by a user with known mechanical ideas in a database, and uses this to shape the structure of a game. This makes it a powerful tool for assistive design, being able to combine knowledge about a real-world concept with specialised knowledge about game mechanics and their meanings. However, in all of the work we have discussed so far, both the source concept maps and the target game mechanics are designed and annotated by people. Removing either of these components makes the task much more complex.

In (Cook et al. 2013) we introduce Mechanic Miner, a system which modifies code using metaprogramming techniques to invent new game mechanics. In the paper we argue that a system that can generate mechanics through code offers a greater chance for novelty and surprise compared to existing grammar-based or abstracted approaches. In (Cook, Colton, and Gow 2013) we provide more discussion of Mechanic Miner and the challenges surrounding code generation for creative purposes. In particular, we discuss what we refer to as *the generation gap* – that is, the disconnect between the functionality of what is generated (e.g. some code that modifies a variable called *health*) and the metaphorical interpretation of what the code appears to do within the context of the game (e.g. the hero is injured by a dragon).

It should be noted that the term *context* when applied to the medium of videogames can refer to many different components of a game. Contextual information can be entirely surface-level – such as the backstories of fictional characters – and have little or no tangible role within the game's functionality. In this paper we are interested in how the program code describing a game's mechanics is related to the contextual meaning of a game. As we saw in the example of *Kaboom*, this is primarily achieved by expressing mappings between nouns and verbs in the real world, and objects and interactions in the game's code. Indeed, the word *verb* is common parlance among videogame designers, referring to a means of interaction with the game world.

To highlight the problem of mapping real-world context to a piece of code, consider the following segment:

```
if(Keys.isPressed("spacebar"){
    for(Agent a : level.agents){
        a.destroy();
    }
}
```

This code describes functionality similar to a simplified *smartbomb*, an item in the arcade game *Defender* (Jarvis and DeMar 1980). When used, all the enemies in the area are destroyed. This makes sense within Defender's metaphorical context - you are controlling a spaceship outfitted with futuristic weapons, fighting against enemy spaceships. By contrast, such a code segment would not be appropriate in a game such as *Versu* (Evans and Short 2013), an interactive fiction game about social conventions in Jane Austen novels. As with Treanor's rebranding of *Kaboom*, by changing the instantial assets of the game which the code operates on, we shift the meaning of the code – in this latter case, to a meaning which is nonsensical bordering on meaning*less*.

To a system generating code, however, the metaphorical concepts are not visible. Relationships that are clear in data – such as commonalities in code, colocation of objects within a co-ordinate system, and so on – do not alone encode the contextual information that separates futuristic spacewar from Edwardian period drama. This complicates the process of generating code for use within games – without an awareness of this context, we cannot robustly and confidently generate code without heavy filtering and guidance from human designers, which is not always practical (in the case of mixed-initiative systems that may be collaborating with people who cannot program) or desirable (in the case of autonomous game designers where the aim is to develop games without human intervention).

In this paper we outline several possible ways in which code generation systems might be able to reason about or glean information on the context the game is operating within. Some of these proposed techniques are simple to implement, but do not entirely solve the problem in a robust and human-free manner. Others are more speculative, but offer longer-term goals for systems that aim to generate code for creative use within videogames.

Extracting Context

In most examples of content generation, the systems involved are not working from scratch. Mixed-initiative systems work in conjunction with people, and therefore receive initial content from their users. Many autonomous systems work from a base of existing code, or have access to lower-level libraries that describe a generic game engine. In such cases, context already exists embedded into the content generated by people. If we can build our systems in such a way that they are able to understand and extract this pre-existing context, we can use it to detect existing meanings in games, and guide the production of content along the same lines.

Implicit Meaning In Naming And Typing

While we stated in the introduction that code is devoid of context, human-authored code *can* carry some forms of context at the level of symbols and syntax. While the *functionality and structure* of code is agnostic to context, many programming language practices are developed to embed information and improve readability. For instance, variable and method names have established patterns and often imply specific real-world features as a result. Consider the following variable declaration from a core game class in the Flixel game library:

```
public class FlxSprite(){
    public boolean alive;
    //...
}
```

The variable `alive` is named specifically to be descriptive and informative. The types and the names of variables include information that hint at contextual scenarios that are expected to arise. Through the use of knowledge databases such as WordNet (Miller 1995) and ConceptNet (Liu and Singh 2004), an automatic system analysing this code may be able to make certain inferences about the game state. For example, Boolean types such as `alive` give us an understanding of the states that this variable can be in with relation to the target object. WordNet provides us with antonyms, suggesting that a `false` value for this field may imply the target object is 'dead'. We can take this analysis further by using SentiWordNet (Baccianella, Esuli, and Sebastiani 2010) to infer which of the two states, if any, has a positive affect associated with it. SentiWordNet suggests that *alive* is a positive term, while *dead* is overwhelmingly negative.

From a boolean variable with a single name we have now extrapolated two states, with definitions and associated sentiments. In order to understand how this variable relates to the systems and objects within the game, we can now search for code segments that alter this state in the player object and, using the assumption that the player wishes to be in positive-affect states, begin to build up a graph of positive and negative relationships between objects.

This approach offers interesting challenges in computational linguistics. Some naming conventions or types of language are used to describe the architecture of game projects rather than the specific game's setting itself. A class named `Agent` might refer to an NPC in a spy game or it may simply be a generic term for an intelligent entity in a simulation. Understanding what data is exposed as surface-level theming (such as the number of gold coins collected) as opposed to data which is only used internally (such as the game's co-ordinate system which the player may not be aware of) is also important. The ability to analyse arbitrary codebases and make such judgements will be a difficult task, and may require a large-scale analysis of many game codebases to try and extract common linguistic patterns that are unrelated to the game's context.

State Change Detection Via Model Checking

Model checkers (Clarke, Emerson, and Sistla 1986) are tools which are typically employed to check that a piece of code meets a written specification. By expressing a condition that we wish to investigate, a model checker can not only verify if it is possible to reach this state, but can show us which code segments are executed in achieving this, and what inputs may cause this to happen. Such tools could be employed to detect relationships built into the game codebase that are hard to detect merely by observing code, but may be obvious when executing it. Consider the following code blocks:

```
//... in class GameState
for(Agent a : level.agents){
    if(a.collidesWith(player)){
        player.kill();
    }
}
//... in class Player
public void kill(){
    this.alive = false;
}
```

Using model checkers, we can analyse this code to find specific program traces that cause certain segments of code to be executed, such as the assignment of `alive` to `false`. In doing so, a system may be able to infer that the `Agent` objects in the list `level.agents`, as well as the `player` object, are all involved in the trace of changing the player's `alive` state. While this set of objects is likely to be extremely large for some codebases and variable changes, analysis over multiple affectively-named variables such as `alive` may allow such a system to infer antagonistic relationships between, for instance, `Agent` objects and `Player` objects. This is enough to begin building a model of interactions between objects, and interactions that are legal within the game's metaphorical environment (such as an object being capable of dying).

Explicit Annotation

An alternative approach to extracting meaning from human-authored code is the use of metadata written with the explicit intention of being read by an automated system. In Java, for example, *annotations* can be defined and attached to language abstractions such as field and methods. Consider the following variable declaration:

```
@verbal{"speed"}
@heightened{"fast"}
@lowered{"slow"}
float velocity;
```

The lines beginning with `@...` can be read by preprocessors or Java itself at runtime, and parameterised data can also be extracted. Here, we have defined three annotations for the field `velocity`. The first is a straight synonym for the field's name, in more natural English that might suit the tone of the game better. The second and third annotations are specific descriptors for higher and lower states of the variable. This is crucial in disambiguating the use of the variable, and helps the system make sense of situations in which the variable is modified. Consider the variable `health`, referring to the number of health points the player has remaining. The variable itself carries no particular affective meaning, so approaches described in the previous sections would not apply. However, increases or decreases to the variable describe healing and damage respectively, which are highly affective situations, and thus important for the system to recognise.

This requires large amounts of initial input from designers or programmers, yet the resulting system is more robust and carries additional information in a subtle and simple way (relying only on single words). This may be more appropriate for larger development projects that rely on large codebases and teams. This approach is unlikely to provide rich contextual information in isolation, however, as single words or phrases may not be enough to convey information about the kinds of systems that are at work, or what the player's motivation is (both enemies and the player have health variables, but the player isn't interested in increasing enemy health). Therefore, this approach may be best combined with other approaches from this section.

Relating Context To Code

In the previous section, we described potential methods for automatically extracting existing context from code. In many situations we may not have existing context, however, or we may wish to invent entirely new contextualisations of a game (such as Treanor's retelling of *Kaboom*). In this case, we need to design methods by which a system can proactively apply context to code, either by inventing entirely new applications of meaning, or by applying abstract knowledge about the world to the game's existing structure.

Established Translations of Reality

Although the task of completely specifying real-world systems is too great, it may be possible to partially specify incomplete models of the world that can be extrapolated to cover combinations of systems or continuous data. A good example here is to consider the standard two-dimensional platform game and how it represents real-world physics. Objects have velocity, acceleration, gravity, position in world space and so forth. This can be built into an abstract model, with annotations describing processes that change the state of an object. For example, we might express a relationship between velocity and a change in position, relating it to words such as 'movement' or 'jumping' as we did with explicit annotation.

The benefit of an abstract model that is separate from a codebase is that it is implementation-independent, and can concentrate on expressing more interesting relationships between objects without worrying about the finer details of code that may obfuscate these more important meanings. For instance, the notion of being alone in a game is expressed as an object that is not near other objects (or perhaps only objects of certain types). Notions of closeness, relative distance, and affective states related to these, can be simply described and related to game state (such as co-ordinates or ranges for variables) rather than referring to specific code blocks or input chains.

For the purposes of autonomous game design or computational creativity, this solution is considerably less desirable than others. The information given is highly prescriptive, and while systems may be able to elaborate on the information given to them creatively, it is unlikely that they will be able to use this information to extrapolate new relationships about the world. This limits the chance for novelty or surprise in the system. However, an abstracted model of a game scenario may be useful for mixed-initiative tools where both human and machine benefit from considering the game design at a higher level of abstraction. It also eases linguistic annotation of game states, and may help systems describe mechanical systems within games. A large proportion of code in an average game is written to achieve subgoals within a larger algorithm rather than meaningfully making high-level changes to the game state. It is unhelpful to focus on every detail of a block of code, as much of it will be unrelated to describing the overall effect of the current action being executed. By building an abstract model to define only that which we are interested in describing, the task of generating text to describe a process is simplified.

Assertive Creativity

So far, all of our proposed methods have assumed that the goal of the system in question is either to extract context and meaning from existing code, or apply meaning to generated code in a way that conforms to models of reality and an understanding of metaphor in videogames. In this section we propose an alternative approach wherein the system proactively assigns both instantial assets and theming onto in-game mechanical relationships that may not make sense in accordance with expected 'game logic'.

Autonomous game designers as independently creative systems has arisen as a topic in computational creativity multiple times, such as (Smith and Mateas 2011) or (Cook, Colton, and Gow 2013). The systems described, hypothetical or otherwise, are intentioned, autonomous systems that create games in a way that is as independent as possible. While the games produced by automated design systems may be abstract and symbolic, it is likely that many of the games designed by such systems will be situated within a real-world or fictitious metaphoric context. It is assumed that this context must make sense from a human perspective, as all existing examples of non-abstract game design are designed from such a perspective.

However, there is no reason why this should necessarily be the case. Some proposed methodologies for evaluating creative systems prioritise the *process* above the *artefact*, such as the FACE model described in (Colton, Charnley, and Pease 2011). For a system attempting to create a context within which to situate an abstract game design or piece of generated code, we might therefore argue that the process by which the context is generated is more important for the system's overall perceived creativity than whether or not the context makes sense to human observers.

We can therefore imagine an autonomous game design system which generates code and *assertively* assigns meaning and context to it, regardless of its agreement with generally-accepted commonsense knowledge. To illustrate what we mean by this, consider a proposed game design, where the player must collect static blue objects spread across a game world, while avoiding collisions with moving red and yellow objects. A context for this game might assign meaningful labels to the yellow, red and blue objects, as well as the player. One assignment that might make sense in the context of the real world is that the player is a bear, collecting honey pots while avoiding bees and hunters. This correlates with metaphors that we understand in videogames (colliding with items is analogous to picking those items up, for instance) as well as cultural knowledge we have about real-world concepts like bears and honey.

If we were to consider a system which asserts meaning independently, it might propose a different contextual assignment – one which is not grounded in a metaphoric understanding of videogames, but has some other justification internal to the system. This justification may be something that is difficult to interpret as an *artefact*, but that is acceptably creative as a *process* of creating context. For some already-established relationship between concepts, perhaps extracted from a news article as in (Cook and Colton 2012), the system may make assignments to game objects where relations have an internally consistent (but potentially externally nonsensical) mapping.

For example, both yellow and red objects have similar effects on the player when they collide. A conceptual mapping might match *causes political unrest* to this code segment, and assigns *rebel forces* and *civilian protests* to red and yellow objects, with the player as a country. This may not make sense from the perspective of a person seeing objects collide in a game world, because collisions imply some physical and spatial relationship. But it is internally consistent in that similar code segments are mapped to similar concepts, and the agents in these relationships are consistent with the concept map the system initially formulated.

This approach to context generation is appealing because it frees the autonomous system from the burden of conforming to accepted videogame context and metaphor, while retaining its strength from a computational creativity perspective. However, it is unclear how this would fare under other evaluatory methods, particularly whether evaluations of the game by people playing them would result in acceptance of the generated contexts, or the dismissal of them as nonsensical or 'random'. In the absence of other ways in which to connect theme to mechanic, however, proactive assignment of meaning may be an appealing first step that allows us to experiment with meaning assignment without being derailed by the difficulty of connecting known videogame metaphors (such as collisions in a physical space representing damage) with real-world concepts.

Conclusions

We have presented a variety of possible solutions to the problem of connecting generated or given code to a generated or given metaphorical context within a game design. We explored approaches that may work both for existing, human-authored codebases, as well as those which are more general and may be able to work with code that has been generated by a machine with no explicit context pre-existing.

We are currently working on a system that aims to generate code and assign meaning to it autonomously, in a greater scope and a finer level of detail than the system described in (Cook et al. 2013). This system will need to communicate the functionality of the code it generates, as well as relate it to some real-world context. Although none of the systems we outline in this paper are currently tested or implemented, we plan to explore some of them soon.

Bridging the gap between context-free program code and a rich, meaningful game theme is a difficult task for many areas of research besides computational creativity. The ability to connect what is seen on a computer monitor to processes and events we have witnessed in real life is a subtle but ultimately commonplace activity for many people today, which increases the difficulty researchers face when trying to build systems which tackle this problem, and when subsequently trying to convince people that the system is doing this. With an abundance of work now emerging both on the side of game studies and within procedural generation communities, however, it seems we are ready to begin building systems that confront this problem head-on.

Acknowledgments

We would like to thank Tony Veale for discussions about linguistic annotations, and Adam Smith for discussion of model checkers as a means of tracing program states. We would also like to thank the reviewer comments that greatly improved the flow and clarity of the paper. Thanks also to Mike Treanor for suggestions that broadened the base of this paper. This work has been supported by an EPSRC grant.

References

Baccianella, S.; Esuli, A.; and Sebastiani, F. 2010. Sentiwordnet 3.0: An enhanced lexical resource for sentiment analysis and opinion mining. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*.

Clarke, E. M.; Emerson, E. A.; and Sistla, A. P. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems* 8:244–263.

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*.

Cook, M., and Colton, S. 2012. Initial results from cooperative co-evolution for automated platformer design. In *Proceedings of the 15th European Conference on the Applications of Evolutionary Computation*.

Cook, M.; Colton, S.; Raad, A.; and Gow, J. 2013. Mechanic miner: Reflection-driven game mechanic discovery and level design. In *Proceedings of the 16th European Conference on the Applications of Evolutionary Computation*.

Cook, M.; Colton, S.; and Gow, J. 2013. Nobody's a critic: On the evaluation of creative code generators. In *Proceedings of the 4th International Conference on Computational Creativity*.

Evans, R., and Short, E. 2013. Versu: A simulationist interactive drama. http://www.versu.com/.

Jarvis, E., and DeMar, L. 1980. Defender.

Liu, H., and Singh, P. 2004. Conceptnet: A practical commonsense reasoning toolkit. *BT Technology Journal* 22:211–226.

Miller, G. A. 1995. Wordnet: A lexical database for English. *Communications of the ACM* 38:39–41.

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

Nelson, M. J., and Mateas, M. 2008. An interactive game-design assistant. In *Proceedings of the International Conference on Intelligent User Interfaces*.

Smith, A. M., and Mateas, M. 2011. Knowledge-level creativity in game design. In *Proceedings of the Second International Conference on Computational Creativity*.

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

Treanor, M.; Schweizer, B.; Bogost, I.; and Mateas, M. 2012b. The micro-rhetorics of Game-o-Matic. In *Foundations of Digital Games*, 18–25. ACM.

Treanor, M. 2013. *Investigating Procedural Expression And Interpretation In Videogames*. Ph.D. Dissertation, University of California, Santa Cruz.