Mechanic Miner: Reflection-Driven Game Mechanic Discovery and Level Design

Michael Cook, Simon Colton, Azalea Raad, and Jeremy Gow

Computational Creativity Group
Imperial College, London

Abstract. We introduce Mechanic Miner, an evolutionary system for discovering simple two-state game mechanics for puzzle platform games. We demonstrate how a reflection-driven generation technique can use a simulation of gameplay to select good mechanics, and how the simulation-driven process can be inverted to produce challenging levels specific to a generated mechanic. We give examples of levels and mechanics generated by the system, summarise a small pilot study conducted with example levels and mechanics, and point to further applications of the technique, including applications to automated game design.

Keywords: game mechanics, automated game design, platform games

1 Introduction

Procedural content generation (PCG) is a highly active area of research, both in academia and game development. The ability to generate high-quality content on-demand and in large volumes not only can improve a game's quality, but has made possible new types of game dependent on their ability to generate content in this way - such as Betts' *In Ruins* or Smith's *Endless Web* [1]. However, much PCG focuses on the generation of consumable data, such as terrain, quests, items or narrative. In order to explore the concept of fully-automated game design, we must find techniques for generating all aspects of a game, including higher-level concepts that describe how game systems interact with one another; and in particular how they interact with the player.

Game mechanics are an important type of player-game interactions; they range from the well-known (such as jumping in a platform game) to the experimental (such as in [2], where the player's movement was controlled by shouting loudly or softly into a microphone). Novel mechanics are still a major source of interest in the independent development community, with many events focusing on mechanics[1] and many awards rewarding innovative design[2]. One approach to generating game mechanics is to compose them out of smaller rules or concepts defined by hand. However, doing this explores a restricted search space, and limits the system's potential for novelty or surprise. Reducing the use of predefined rules or domain knowledge may help mitigate this.

[1] e.g. the Experimental Gameplay Project (http://experimentalgameplay.com/)
[2] e.g. the Independent Games Festival's *Nuovo Award.*

We introduce here Mechanic Miner, an evolutionary system designed to generate simple game mechanics, and then design levels that specifically require the use of those mechanics in their solution. We show how the reflective properties of a programming language can be used to generate mechanics programmatically without metalevel domain knowledge, and demonstrate a simulation-driven approach to evaluating mechanics for utility. We then show how, by inverting the evolutionary system, we can use the same process and principles to design levels for specific mechanics, with control over features like difficulty and complexity.

The rest of the paper is organised as follows: in section 2 we describe the process of evolving game mechanics through simulation; in section 3 we show how this process can be reversed to design levels that use certain mechanics; in section 4 we give examples of generated content and describe a small pilot study; in section 5 we review related work to this project and distinguish our approach and in section 6 we conclude and describe some directions for future work.

2 Automatic Generation of Game Mechanics

2.1 Background

Reflection Reflection is the ability of a programming language to inspect itself at runtime, allowing for the runtime creation of new classes, the modification of code, and the inspection, invocation and alteration of fields and methods. The following code retrieves a list of objects describing the fields of an object o:

```
Class<?> c = o.getClass();  List<Field> fs = c.getFields();
```

Mechanic Miner is further extended by the open source Reflections library[3] in order to overcome the limitations of Java's standard reflection.

Toggleable Game Mechanics In the remainder of this paper we refer to *toggleable game mechanics* (TGM), an intentionally simplified subspace of game mechanics that we identify for the purposes of demonstrating Mechanic Miner. A TGM is an action the player can take to change the state of a variable. That is, given a field foo and a modification function m with inverse m^{-1}, a TGM is an action the player can take which applies $m(foo)$ when pressed the first time, and $m^{-1}(foo)$ when pressed a second time. The action may not be perfectly reversible; if foo is changed elsewhere in the code between the player taking actions m and m^{-1}, the inverse may not set foo back to the value it had when m was applied to it. For instance, if foo is the player's x co-ordinate, the and the player moves around after applying m, then their x co-ordinate will not return to its original value after applying $m{-}1$, as it was modified by the player moving.

Flixel and Flixel-Android Flixel[4] is a popular open-source game library built on top of Actionscript 3. Flixel-Android[5] is a port of the Flixel library to the

[3] http://code.google.com/p/reflections/
[4] http://www.flixel.org
[5] http://code.google.com/p/flixel-android/

LibGDX framework. It follows the same structure and has the same method calls (with some concessions made for differences in programming language capabilities). LibGDX[6] is a Java library that compiles to many platforms.

2.2 Mechanic Miner

Mechanic Miner is an evolutionary system for searching a codebase for *usable* TGMs. The codebase is defined as any code in the game being analysed – for our experiments, this means both the Flixel-Android library code, and the code of a simple platform game written using the library. We define a *usable* mechanic as one which enables the player to overcome some obstacle in order to complete a task, such as reaching an exit. Figure 1 shows a sample level where the player's progress to the exit, marked 'X', is blocked by a tower of impassable blocks. The player starts in the square marked 'S', falls to the ground under gravity, and is unable to jump over the tower. A usable TGM is any TGM that allows them to reach the exit from this starting configuration. One example would be a mechanic that toggled gravity off, allowing the player to jump as high as they wanted temporarily, and thereby leap over the tower in the centre.

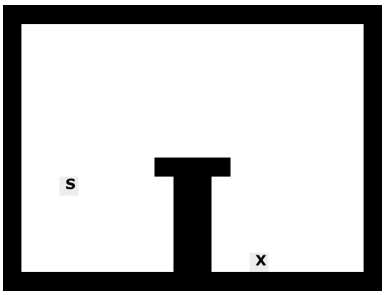

Fig. 1. A sample puzzle level. The player starts at the 'S' mark, and must reach the 'X' mark. Black squares are solid, and gravity acts on the player pulling them down.

Population Generation via Codebase Inspection In section 2.1, we defined a TGM as a combination of a game variable and a function that modifies it. In Mechanic Miner, we formalise this by describing a mechanic as being comprised of two parts: a `java.lang.Field` object that refers to a field present either in the game library's code or the code of the template game; and a `Modifier` that encapsulates an operation on a field to change it. In order to generate such mechanics, we use reflection to search through all the classes defined in the Flixel-Android library and the template game. We choose fields at random, and based on their type we randomly select a modifier to use. The two together comprise a TGM. Modifiers are type-specific because the range of operations that can be performed on a field are restricted by the type system. For numerical types like

[6] https://github.com/libgdx/libgdx

`float` or `integer`, we offer modifiers that double or halve the value, or multiply it by -1. For `boolean` fields, we invert the sign of the boolean. The set of modifiers we use is limited for this work to simplify the system.

Fitness Evaluation via Simulation We evaluate the fitness of a mechanic during evolution by simulating its use in the sample level, testing its usability, and calculate its score based on the area the player gains access to. The process of simulation is intended to find all valid sequences of *actions* under a certain length, using a breadth first search. An action is a set of steps that the simulated character performs until it can no longer reach any new space. For instance, the action `MOVE` encapsulates moving left or right, until all areas reached by only moving left or right have been found.

The breadth-first search process simulates one action at a time, and for each level location reached by the character the simulator checks a data structure for that location to see if the same (or shorter) sequence of actions has already reached that location. If not, the sequence is new for this co-ordinate, and so it adds a new node to the search space to be explored. Because this node is one move longer than the node that was just explored, the breadth-first search will only expand it after it has finished searching all the nodes with shorter sequences. In this way, we find the shortest sequence that reaches the exit. For the mechanics and levels described in this paper, the actions used were `MOVELEFT`, `MOVERIGHT`, `JUMP`, `SPECIAL` and `NOTHING`. `NOTHING` is an action used when simulating a fall.

When a sequence of moves that reaches the exit location has been found, it is returned instantly (as this proves the TGM is *usable*, under the earlier definition). The fitness of a mechanic in this instance is proportional to the amount of the game world that the player was able to access before reaching the exit. This is a good metric as it allows the system to avoid finding mechanics that are too empowering (allowing the player to reach everywhere in the game world) or not empowering enough (mechanics that, for instance, instantly teleport the player to the exit location, which offer little flexibility for interesting level design).

Crossover, Mutation, and Evolutionary Parameters Mutation of a TGM is performed by either randomly varying the modifier on the mechanic (e.g. changing a `Float`'s modifier from *Halve* to *InvertSign*), or by changing the TGM's target field to another field from the same class within the game engine. Specific mutations are selected randomly from all legal mutations for the mechanic (if a type has only one modifier, for instance, it will not attempt to randomly reassign the modifier as it will result in an identical mechanic). Crossover of two mechanics uses uniform crossover where the two mechanics affect fields of the same type. In the case that the two mechanics do not, mutation takes place instead. 10% of the new population is comprised of newly-generated TGMs.

A standard run of Mechanic Miner maintains a population of 100 mechanics, evolved for 15 generations. Simulation (for fitness evaluation) is limited to ten discrete actions, such as `MOVERIGHT`, before the simulator stops under the assumption that no solution can be found (the simplicity of the sample problem

means that it is unlikely that solutions longer than ten moves will be found - none found during experimentation took more than ten moves). These parameters were all determined through experimentation with the system.

3 Mechanic-Led Level Design

Once a TGM has been identified by Mechanic Miner as potentially usable for the candidate problem, we can use it as the basis for designing levels that specifically require the use of that mechanic by players en route to finishing the level. This section outlines the process of evolving level designs for specific mechanics.

Population Generation A level is described abstractly as a collection of geometric shapes that prescribe the placement of impassable blocks, similar to the system described in [9]. Each shape is either a *Line* or a *Box*. Boxes may be filled or outlines only, and Lines may be standard blocks, or spikes which kill the player on contact. The generation of a level consists of the random placement of shapes, with a limit on the minimum and maximum number of shapes a level can contain. For the levels generated in this paper, they contain no fewer than 2 elements and no more than 20.

Simulation-Driven Fitness Evaluation In order to evaluate the level, we simulate multiple attempts at solving it, using the simulation method outlined in the previous section. The first simulation does not use any mechanics, to test if the level can be completed without mechanic use at all. If this is the case, the level receives a penalty to its fitness, as we want to end up with levels that can only be solved using the new mechanic. The second simulation is a standard simulation using the mechanic found by Mechanic Miner. A third simulation can also be run in order to evaluate the difficulty of the level. In this simulation, the mechanic is used but the amount of time processed between each simulation step is increased. By increasing the time the game engine runs between actions by the simulator, we can simulate the player having slower reaction times. This makes it possible to approximately parameterise the difficulty of a level, by penalising the fitness of levels that can be solved with long reaction times.

When the three simulations are complete, assuming the level was not completable without mechanic use, the fitness is scored as follows: first, the length of the shortest successful trace is compared to the target trace length, TTL. We assume that longer traces, implying more complex chains of action, imply a harder puzzle to solve (since we are agnostic to the mechanic being designed for, this seems a reasonable assumption). Levels whose shortest trace is closer to the target trace length receive a higher fitness. This accounts for half of the total fitness score, normalised between 0 and 0.5, where 0.5 represents a level whose shortest trace is as long as the target trace length. We also compare the number of times the mechanic was used with a target mechanic use variable, TMU, using the logic that frequent use of the mechanic implies a complex solution. This is

also normalised between 0 and 0.5. The variables TTL and TTM allow us to adjust both the overall complexity of the level's solution, as well as the ratio of mechanic use to other actions.

Crossover, Mutation, and Evolutionary Parameters Mutating a level involves the random replacement of abstract level elements with new elements, or the adjustment of parameters such as the length of the lines and box sides, or their starting co-ordinates. We employ uniform crossover by selecting a point in the level grid, and performing crossover of any level elements whose starting co-ordinate lies before that point on the grid. Parameters vary depending on the desired difficulty of the level being designed, which is adjusted by the user through the target trace length and other parameters described in the previous section. A typical run maintains a population of 100 levels, run for 15 generations, with a target mechanic use of 2 and a target trace length of 6. The standard target reaction time is a step of 64ms, with a slower reaction time of 256ms.

4 Results and Evaluation

This section includes three game mechanics, and three levels designed for each, which require the use of the associated mechanic in order to be solved. The player starts in the 'S' position and must reach the exit, marked 'X'. Red tiles kill the player, while black squares are solid ground.

4.1 Mechanic - Gravity Inversion

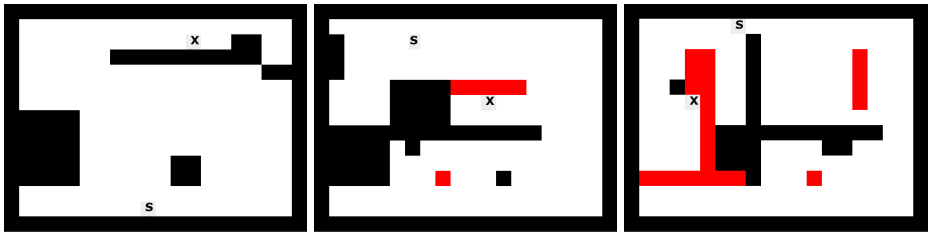

Fig. 2. Gravity inversion mechanic: `INVERTSIGN player.acceleration.y;`

In Flixel, setting an object's acceleration field is a way to simulate gravity, so a typical platform game normally sets the `acceleration.y` field to a positive integer. Multiplying this value by -1 has the effect of inverting gravity. Figure 2 shows three sample levels.

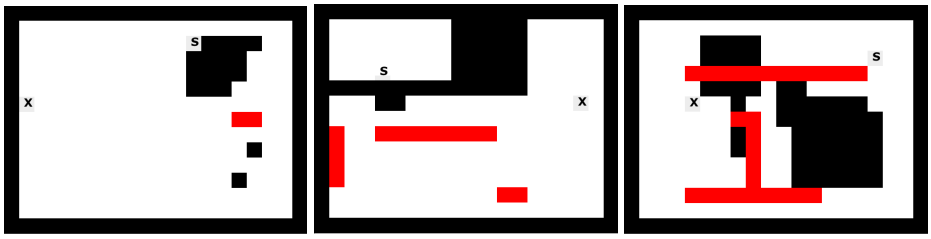

Fig. 3. Teleportation mechanic: `HALVE player.y`

4.2 Mechanic - 'Teleportation'

Halving (doubling, if activated again) the player's y co-ordinate allows for teleportation around the game world. This mechanic is unusual, as the classic mental model of a platform game does not match up to the co-ordinate model the game engine uses. Here, (0,0) refers to the top-left corner of the screen. Halving the distance between the player's position and the ceiling has no natural analogue in real-world physics. This makes the mechanic technically usable, but confusing to the player. Figure 3 shows three sample levels.

4.3 Mechanic - 'Bounce'

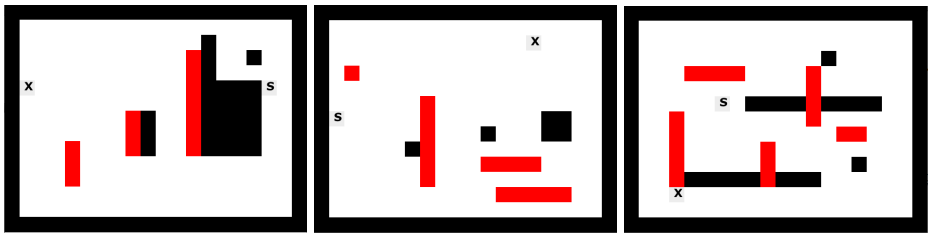

Fig. 4. Bounce mechanic: `ADD 1 player.elasticity;`

Elasticity affects how an object interacts with the game's physics engine. Elasticity is normally set to 0; any positive number will cause the player to bounce when it collides with things. This mechanic allows the player to become 'bouncy' on demand, gaining momentum to bounce over larger gaps or to greater heights. Figure 4 shows three sample levels.

4.4 Pilot Study

We performed a small pilot study to gain insight into how some of the mechanics and levels generated would be received by human players. 7 participants with a

variety of experience with videogames took part in the study, which consisted of twelve levels split into sets of three sharing a common mechanic. The levels were designed with variable evolutionary parameters; in particular, we attempted to present progressively harder levels to the participants by increasing parameters such as overall solution length and reflex requirements. The mechanics presented to the player were those described above, and a fourth mechanic that allowed the player to halve the effect of gravity. The participants were not told how the mechanics worked, beyond being told they had a special ability they could use.

When asked to describe how they thought the mechanics worked, all five participants failed to describe the teleportation mechanic accurately, and all rated it as unenjoyable. This supports our belief that some mechanics invented by Mechanic Miner in its current form are confusing for players, because they may refer to elements of the game code that are not in the player's mental model of how the game world works. By contrast, all of the participants rated the gravity inversion mechanic highly, with the hardest level of the set described as "impressive" and "fun" by two of the participants. Comprehension of a mechanic seems understandably integral to a player's enjoyment of using it, and the feedback from the study raises questions as to whether pure utility may be an effective evaluation criteria for a reflection-driven system. Participants were also asked to rank the levels in order of perceived difficulty for each set of three levels. From the responses received, it seems that our metrics (such as the number of times a mechanic must be used in a solution) model difficulty well for some mechanics, but poorly for others - participants tended to agree on which levels were harder than others, but in the case of mechanics such as bouncing, it did not match up to our intended difficulty ordering. This may imply that different mechanics require different metrics, as some may be suited to reaction challenges, while others may be about ordering of actions or puzzle solving.

The study has informed the design of a larger survey that will focus on a subset of mechanics. We hope this will provide more evidence for our hypotheses.

5 Related Work

Our related work covers both ruleset and mechanic invention. A common approach is to select mechanics from an annotated database. The system described in [3] uses a list of known mechanics and game content, associated with words, and uses WordNet to connect input strings to words attached to known game content. This allows for the verb *shoot* to connect to mechanics where the player controls a crosshair shooting at objects, for example. The work in [4] uses smaller mechanical pieces which the authors call *micro-rhetorics* (see [5]). The Game-O-Matic [4] takes sets of these micro-rhetorics and matches them with a human-specified network of entity relationships. In one example in the paper, the relationship *dog needs food* is expressed through two objects labelled 'dog' and 'food', with the player being able to control the dog, tasked with gathering food.

These annotated-database approaches work well where a human designer is able to construct a database of known concepts beforehand, but this limits the

autonomy of the system, as well as its ability to surprise us through novel output. This makes it a promising direction for mixed-initiative tasks. However, we are interested in building an autonomous game designer, from a Computational Creativity perspective [6], and this approach is less suited to our interests.

The other main approach to mechanic design is a grammar-based constructive approach that uses templates, into which small sub-mechanic components are inserted to construct high-level rules [7, 8]. Here, rule templates describe abstract notions of mechanics (such as an event where two objects collide and effects are applied to them) and then abstract notions of smaller objects/events/effects that are in common videogame vocabulary (game objects such as the player, moving entities, or the game level wall; sub-events such as an object being killed, or teleported to a new location). This allows for novel composite mechanics to be constructed, but the essential elements of the rules still direct the system towards rediscovering known concepts or slightly elaborating on them. The need for a technique that is not dependent on prior definitions is still evident.

6 Conclusions and Future Work

In this paper we introduced Mechanic Miner, an evolutionary system that generates new game mechanics through Reflection and then validates their usefulness by simulating game playouts. We also showed how the same techniques can be used to design levels that can only be solved using the mechanics invented. Mechanic Miner represents a novel method for both discovering potential game mechanics and exploiting them through level design, without knowing anything about the game engine or the mechanics found during the search. We have found the system to be capable of producing mechanics unexpected to us (such as the mechanic in section 4.3) as well as discovering mechanics similar to those used by human game designers ([11] uses a mechanic similar to 4.1).

The simulation-driven level design system also surprised us by discovering exploits within the game code we had supplied it. The sample game includes a check that the player is touching the floor before allowing them to jump. Mechanic Miner found that by using a teleportation mechanic it could teleport inside a solid wall, which Flixel registers as the same as touching the floor, effectively allowing the player to 'wall jump'. Exploiting game engines in such a way is common in certain game communities, and integral to competitive 'speed runs' of games, where players try to find new ways to shortcut or break the rules of a game. That Mechanic Miner was able to do this, without any anticipation from the system's authors, is an exciting indication that the system may be able to find more complex, emergent game mechanics in future.

Some of the areas suggested for possible future work include:

Game Object Synthesis Rich game mechanics often affect in-game objects that are created as extensions of the game engine. For instance, some of the mechanics found by Mechanic Miner affect the Player object, which extends Flixel's basic FlxSprite object. By allowing Mechanic Miner to create new types of game object as part of its search process, and then find mechanics that exploit

interactions between the new game objects and the game world, we open up more interesting possibilities for discovery.

Richer Reflection and Constraint Generation Some mechanics only make sense in the context of complementary vulnerability. Mechanic Miner discovered the notion of gravity inversion, which is used in [11]. However, in [11] the player is unable to jump. This simple adjustment allows for many new challenges to be designed. Constraints are just one way that reflection could be leveraged to create more intricate mechanics. Reflection can be used to call methods or run arbitrary code at runtime, opening up the possibility for more complex, multi-operation mechanics to be constructed beyond field modification.

Emergent Mechanics In some games, problems are presented to the player that cannot be solved with a single mechanic, but can be completed using a combination two. By modifying Mechanic Miner, we believe it can be used to solve levels using multiple mechanics at a time, potentially discovering scenarios where mechanics can combine to solve new types of problem.

7 Acknowledgements

Thanks to Julian Benson for discussion of his work on player traces in *Braid*, and Julian Togelius for input into the Mechanic Miner evaluation. Thanks also to the anonymous reviewers for some insightful comments.

References

1. G. Smith, A. Othenin-Girard, J. Whitehead, and N. Wardrip-Fruin, "PCG-based game design: creating endless web," in *Proc. of the International Conference on the Foundations of Digital Games* (FDG), 2012.
2. *GNILLEY*, Radix, 2010. http://www.gnilley.com/
3. M. J. Nelson and M. Mateas, "Towards automated game design," in *Proc. of the 10th Congress of the Italian Association for Artificial Intelligence*, 2007.
4. M. Treanor, B. Blackford, M. Mateas, and I. Bogost, "Game-o-matic: Generating videogames that represent ideas," in *Proc. of the Third Workshop on Procedural Content Generation in Games*, FDG 2012.
5. M. Treanor, B. Schweizer, I. Bogost, and M. Mateas, "The micro-rhetorics of game-o-matic," in *Proc. of the International Conference on the Foundations of Digital Games* (FDG), 2012.
6. S. Colton and G. Wiggins, "Computational creativity: The final frontier?" in *Proc. of the 21st European Conference on Artificial Intelligence*, 2012.
7. J. Togelius and J. Schmidhuber, "An experiment in automatic game design," in *Proc. of the IEEE Conference on Computational Intelligence and Games*, 2008.
8. M. Cook and S. Colton, "Multi-faceted evolution of simple arcade games," in *Proc. of the IEEE Conference on Computational Intelligence and Games*, 2011.
9. L. Cardamone, G. Yannakakis, J. Togelius, and P. Lanzi, "Evolving interesting maps for a first person shooter," in *Applications of Evolutionary Computation*, Lecture Notes in Computer Science, 2011, vol. 6624
10. *Offspring Fling*, KPULV, 2011. http://offspringfling.com/
11. *VVVVVV*, Terry Cavanaugh, 2010. http://www.thelettervsixtim.es