Proceedings of the Twenty-First AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2025)

Pixie: Code-Level Mechanic Generation for Game Designers

Michael Cook

Human-Centred Computing Group
King's College London
mike@possibilityspace.org

Abstract

Procedural generation has been applied to many kinds of game content including levels, puzzles and narrative. Although PCG researchers have tackled the generation of rules and mechanics, these systems typically operate on bespoke codebases designed for mechanic invention tasks. In this paper we present Pixie, a system for generating and testing game mechanics that can be installed in any Unity project and configured using a simple annotation process to focus its scope and objectives. We describe the system's design and operation, demonstrate its ability to generate mechanics for several open-source Unity games, and provide an initial discussion of developer perspectives on the content it generates and its usefulness as a design companion.

Introduction

Procedural generation has been used to assist with or automate the creation of many different kinds of game content, from forests and particle systems, through to narratives, boss behaviours and even entire game designs (IDV 2002; Mateas and Stern 2003; Butler, Siu, and Zook 2017; Summerville et al. 2018). Many of the most popular games of the 21st Century incorporate procedural generation in some way, including Minecraft, the best-selling videogame of all time, and Dwarf Fortress, which along with Minecraft has been acquired by the Museum of Modern Art in New York (Mojang 2009; Adams and Adams 2006). Procedural generation is a crucial tool for artists, designers and engineers, and is applied in many different ways.

Research into new techniques for generating and evaluating game content is also flourishing, but the application of these new ideas is unevenly distributed across different parts of the game development process. One area which has seen little application is in the design of game systems, mechanics or rulesets – what we might consider the 'logic' of the game that governs how different elements respond to the player, or each other, during play.

One limitation on the use of AI in this context is that most videogames are unique in some way, from the high-level conceptual space they work in to the low-level implementation of individual features. Other kinds of game content are less affected by this because they are abstracted assets or use portable data formats. For example, SpeedTree is an industry standard procedural modelling tool for trees and plants, used in a variety of AAA games (IDV 2002). It exports to file formats including FBX and OBJ, which are supported by most industry standard tools, game engines and graphics pipelines. This allows generative tools like SpeedTree to be developed independently from any specific game design or engine, and then connected to a development pipeline.

Some game content exists in a grey area between universal compatibility and bespoke implementation. Level generation (or any generation of physical spaces where gameplay takes place) is one of the most popular uses of procedural generation, despite the fact that most game spaces or levels will include elements that are unique to a particular game. For example, the first level in Super Mario Bros. includes Goomba enemies, boxes that can be bumped into to reveal powerups, secret pipes that can be entered, all of which have a unique role within the game's design that will not be present in any other game. However, game levels often have shared conceptual cores that can be abstracted out, resulting in techniques (and sometimes tools) that are applicable to many different games (Lindeijer 2008). Many popular techniques for generating levels focus on the physical structure of a world represented as a grid or a graph of connected modules or tiles, and are agnostic about the function of game elements such as enemies, collectibles or objectives.

In this paper, we consider the problem of generating mechanics or systems within a game. While research exists in this space, it has largely focused on bespoke games made as research platforms (Cook et al. 2013; Zook and Riedl 2014; Guzdial et al. 2019). This is important to allow researchers to effectively test techniques in a controlled environment, but it leaves open the question of how game developers can adapt such techniques to their own practice, or how to develop third-party tools that can be easily integrated into an existing game codebase.

We present Pixie, a plugin for the Unity game development environment that can generate, test and evaluate game code, using computational evolution and metaprogramming to generate and test code against a given design objective. Pixie is agnostic to the specific game it is applied to. Instead, we use a lightweight system of annotations, reflection and live code compilation to let the user define how Pixie interfaces with the game, where it inserts generated game

mechanics, and how it executes the game for testing. This allows the user fine-grained control over which parts of the codebase Pixie has access to, what kinds of code snippets it generates, and how it evaluates success. We demonstrate this by showing results of applying Pixie to several open-source games made by other developers, and report on a short survey of game developers to provide an initial sense of how Pixie's outputs are perceived. Finally, we reflect on the implications and limitations of this system, and its position in the current AI landscape.

Background and Related Work

Generation of Game Mechanics

The definition of 'game mechanic' is not exact, and academic attempts to define it have had mixed success. We do not intend to fix an objective definition of the term here, but for the purposes of this paper we consider a game mechanic to be any subsystem within a game which has some in-world impact on the game state, and in turn affects the player's behaviour. Game mechanics are often spoken about as self-contained concepts; for example, planting seeds in *Stardew Valley* is a mechanic. However, mechanics are often grouped together into higher-level systems, in that we might describe Stardew Valley's crop farming as a mechanic as well. For the purposes of this paper, we are interested in smaller-scale mechanics that can be described in tens of lines of code.

Generation of game mechanics has a small but significant history in procedural generation research. Early work by (Nelson and Mateas 2007) in automated game design focused on the recombination of fixed game mechanics, with (Treanor et al. 2012) further exploring this to use fragments of mechanics that could be combined to make mechanics with an intended interpretation. Game mechanic generation has a close connection to automated game design for a few reasons. Culturally, 'game design' is often spoken of as being chiefly concerned with systemic design and the design of mechanics (Cook and Smith 2015). Our prior research into automated game design includes a system for generating one-button game mechanics (Cook et al. 2013), while more recently GEMINI by (Summerville et al. 2018) was a mechanics-first automated game designer that extended Treanor's research into meaning by enabling bidirectional analysis of meanings from games, and games from meanings.

In recent years, researchers have looked to machine learning to augment the generation of game mechanics, primarily for evaluation: (Poldervaart 2023) uses GO-Explore to test generated mechanics, while (Gonzalez, Cooper, and Guzdial 2023) use reinforcement learning, although neither use machine learning for the generation of the mechanics themselves. While not focused on mechanic generation specifically, Anjum et al. have explored the use of ChatGPT to generate parts of a game design, including code for new features (Anjum et al. 2024). This latter example has some caveats that separate it from the focus of our work: the LLM did not have access to the game's codebase, and its suggestions were not aiming to solve a specific design issue, nor were they validated for their impact on the game. Rather, the work aims to explore how useful it is to ask ChatGPT for assistance in implementing ideas. Our approach does not use machine learning for either generation or testing of game mechanics; we discuss this later in the paper.

Research and Game Development

The relationship between games research and modern commercial game development is complicated. In 2020 Lai et al. noted a 'lack of collaboration and understanding between academia and industry' (Lai, Latham, and Leymarie 2020), a state of affairs which has seen some improvement in the intervening five years, but is still far from solved. Tracks for practitioners at events such as AIIDE, startups aimed at translating game research to the industry, and the very largest game studios employing R&D specialists all help bridge the gap. However, translation of research by game developers appears to still be rare, and happen slowly, and 'impact' is often narrowly construed in terms of the biggest commercial studios (Keogh 2023).

In (Cook 2020) we argue that in order for research in automated game design (including mechanic generation) to be properly integrated with modern game development, there must be a consideration of how games are structured as software. In particular, researchers must plan how their research software interfaces with the design of individual game projects, and find ways for developers (who are unlikely to be experts in the research) to easily adapt it. We give examples where small differences in how a basic feature is implemented might radically change what information or affordances are available to an AI game design tool.

We suggested that in the future game developers might need to change their approach to software engineering in order to present interfaces that are more amenable to AI (not LLM) understanding. However, we also acknowledge that in the short-term it would be best for researchers to seek solutions that can fit more easily into the way game developers currently work. This is the approach we take in this paper.

Pixie

Pixie is a plugin written in C# for Unity, one of the most popular game engines in the world with over 1,500,000 game developers using the engine monthly. Nothing about Pixie is specific to the Unity environment, and it could easily be reimplemented for another engine that met Pixie's requirements for metaprogramming and runtime compilation. We chose to use Unity as a way to demonstrate its effectiveness on a popular game engine in use by a diversity of developers – including us. We discuss wider applications of Pixie in *Limitations and Future Work*.

Pixie's uses search algorithms to generate and test code snippets that, when integrated into a game's codebase, maximise a user-defined objective function during gameplay. To achieve this Pixie uses computational evolution, creating a population of code snippets and compiling them into *assemblies*, then adding them to the game at runtime. The code it generates, the locations in the game where the code is called, the scenario it uses for testing and the objective function it is optimising for are all customisable by the user, often without writing any code at all. In this section we will walk through each step of Pixie's generative process.

Code Annotation

To install Pixie the user first adds our codebase to their Unity project. Among other things, our codebase adds several *code annotation* types which the user can apply to their code. Annotations are metaprogramming concepts that can label a piece of code in a way that can be detected programmatically. Unity itself uses annotations to indicate, for example, that a component requires another component to be present. An example from the Unity documentation:

```
1 [RequireComponent(typeof(Rigidbody))]
2 public class PlayerScript :
      MonoBehaviour
```

Pixie defines five annotations. `TargetableMethod`, `TargetableProperty` and `TargetableField` can be applied to methods, properties and fields to explicitly allow Pixie to call and reference them in its generated code. By default, Pixie will not use any part of the game's codebase. This is both a security feature (to avoid code being run or data being modified accidentally) and a design feature, as it allows the user to specifically target Pixie's exploration of the design space. For example:

```
1 //Can be used by Pixie
2 [TargetableField]
3 public float playerSpeed;
4 //Cannot be used (no annotation)
5 public float uniqueObjectId;
```

In addition to this there are two modifier annotations. `Clamp` can be applied to targetable class fields of numeric types and allows the user to specify a range for the value of the field. This is used when creating object specifications; Pixie will initially set the field's value to be within the specified bounds. The other annotation, `FieldExtension`, allows the user to explicitly add access to certain fields-of-fields that should not generally be accessed by Pixie. For example, if we wanted the `x` and `y` components of the player's velocity to be accessible, but not allow access to these fields on `Vector2` types generally, we can add an exception for the player:

```
1 [TargetableField]
2 [FieldExtension("x", "y")
3 public Vector2 velocity;
```

The user can add annotations throughout their codebase to set up an experiment with Pixie. They do not affect the normal execution of the game in any way.

Scenario Configuration

In order to test a generated code snippet, the user must specify where in the codebase it will be called. They do this by adding *hooks* to their codebase wherever they would want the generated code to be run. At runtime it is not possible to dynamically insert code into a script that has already been compiled; however Pixie can write, compile and execute *assemblies* – smaller, self-contained chunks of code – and call them in response to hook invocations. This is how Pixie dynamically swaps code in and out of a codebase at runtime. The hook acts as an abstract interface which allows the game to make calls to code which just been generated. For example, if we want to generate some code to execute when a collision occurs, we can insert a hook:

```
1 public void OnCollisionEnter(Collider2D
      other){
2   hook.callback("hitSomething", new
        object[]{this, other});
```

In the above example, `hook` is a local field to this class, which has registered itself with Pixie. The call passes two arguments: the name of the hook, and an array of objects that act as parameters to the hook. These will match up with specifications the user passes to Pixie, described later in this section. The user can define a hook to take as many or as few objects as they like: more objects provides Pixie with more options for its generated code, which can have advantages and drawbacks.

Finally, the user must configure a test scene in which the code will be executed. This is dependent on the game and the designer's goals, and can range from a full playout of a game session, to what we call a 'flashtest' of a single moment of gameplay. We describe how some of these scenarios are set up later in this paper. The only requirements for the test scene is that it must be able to launch without any other context (e.g. if it requires other scenes to be loaded, it must do so itself), and that running the scene must not modify the game permanently. In all of our example cases, the test scene is simply a real scene from the game, with a small modified script to trigger an automatic playout. We discuss this requirement later in the paper, but an 'automatic playout' can be as complex or as simple as needed.

Evolutionary Setup

So far the user has labelled parts of their codebase for use by Pixie, set up a test level in their game with automatic playouts, and defined points in their code where they call 'hooks' for code generated by Pixie. The next step is to define the evolutionary process by adding two new files to Pixie– a run specification file, and a root file.

Run Specification A run specification allows the user to execute code before and after a playtest, and to describe how to evaluate the outcome of a playtest session. The evaluation function defines the objective that drives Pixie's evolutionary system. Examples of objectives might be the distance between two objects (for example, the player and an exit); a performance metric such as score or how long the player survived; or a higher-level metric unique to the playtests – in one of our later case studies we measure how much the player is forced to move around, encouraging Pixie to design a scenario that creates moving dangers.

Run specifications can be very simple, consisting of just a few lines of code for a score function and nothing else. Methods for setting up and tearing down playtests are useful for running more complex tests. In our *Dodgeball* case

study, each playtest is run six times, and in three of the runs the specification file modifies the playtest agent to play at a lower skill level. This allows us to base a score around the difference between two player archetypes, for example.

Root File The root file is the entry point for running Pixie. At a minimum, it configures Pixie by passing it a run specification, and then passes it a series of *hook specifications* which define the type signatures of the code snippets Pixie will generate. Any hook that is called in the game's codebase but not defined in the root file will be ignored.

The root file can also provide Pixie access to methods and fields that cannot be labelled manually. Unity is a closed-source game engine, meaning we cannot annotate some parts of the engine code, including critical data such as object sizes and positions; and the state of physics objects. The user can explicitly reference these in the root file, essentially acting as a kind of promise to Pixie that such methods and fields exist and can be referenced.

The root file is also used to list 'prefabs', a pre-defined specification for a type of game object that Pixie can create through its code. Since there is no way to 'annotate' a prefab object in Unity and we do not want to force the user to move or duplicate objects in the game's project, we instead simply reference them in the root file and tell Pixie their type so it can instantiate them in its generated code. Below is another snippet from our examples:

```
1  AddPrefab(typeof(Ball), "BreakoutMain.
       _ballPrefab");
```

This tells Pixie that an object of type `Ball` exists to be instantiated, and how to reference it. Examples of root files and run specifications for some of our case studies can be found online, in a Github gist[1].

At this point the user has annotated their code, configured a test scenario, and added setup information into a root file that specifies what code to generate, what data can be used in that code, and provided a run specification that describes how to test the code. Pixie can now be run.

Evolutionary Search

Code Generation Pixie works using a single-population evolutionary search process, where a population member is a set of hooks, or code segments, one for each of the hook specifications defined in the root file. The initial population is randomly generated. To randomly generate a hook, we first collect a *scope* for the hook, based on statically available methods, fields and properties (including user-tagged items, and arguments passed to the hook). We walk over the codebase and use reflection to inspect methods, fields and properties to find annotated examples to add to the scope, and then combine it with any explicitly added references from the root file. In the current version of Pixie hooks are always `void` - they do not return anything.

We then generate a number of lines of code, randomly chosen between pre-set minimum and maximum numbers (2 and 6 used for the examples in this paper). Lines can currently be variable declarations (creating new local variables), method invocations or assignments. This is based on an assessment of game mechanic examples and their implementations, to cover a large space of solutions (with acknowledgement of a need to expand the space in future). Whenever an expression of a particular type is required, Pixie can select from methods and fields in scope, predefined static values and a set of default values to create expressions of that type. For example, if we were to assign a value to an object's `speed` variable, some of the (many) ways we could generate an expression of type Float would be:

- Referencing the static field `Physics2D.gravity` that was added in the root file.
- Referencing the local field `var1` that was declared by Pixie on an earlier line in this hook.
- Invoking the method `Mathf.Min(float, float)` which would trigger two further expressions of type Float to be generated.
- Directly using a prepared primitive value of `1.1f`.

For recursive calls, such as method invocations which require further expressions to be generated, we can put a cap on how many times Pixie is allowed to recurse, which limits the length of lines. We cap this at 2 in our examples, as we found even this number can result in very long lines of code. For predefined default values, we currently have a range of values for each type. These are chosen to provide variety without being overly broad. For example, Float primitives can be one of -10, -1, 0, 0.5, 0.9, 1, 1.1, 2 and 10. We believe this should be customisable by the user, however, and note it as a point of future work. We also believe that giving Pixie the freedom to adjust these specific values as part of a second fine-tuning evolutionary search would be useful for certain games. However, we want to emphasise Pixie as a tool for design space exploration and inspiration rather than fine-tuned optimisation.

Generated code is stored in a semi-abstract data structure called a *code element*, approximately equivalent to a single line of code. This allows us to store metadata about the line of code and will allow us to extend the system in the future, for example to enable swapping parts of a line out more easily during mutation or crossover.

We discuss the benefits and limitations of our approach later in this paper, however the specific mechanisms by which we generate code are not the main contribution of our work. The important aspect of this subsystem is how Pixie's overall design enables it to be easily configured by a user and remain well-behaved: the annotation and reflection system allows the user to carefully prune its scope, while the hook interface ensures the code is only called in an appropriate context.

Computational Evolution Once a population has been generated, Pixie proceeds with the core of the evolutionary algorithm. For each generation, Pixie iterates through the population and for each genotype compiles the hooks' code into assemblies using the C# compiler at runtime. The assemblies are then attached to their hooks so that when the

[1] https://gist.github.com/gamesbyangelina

game calls a hook it executes the newly compiled assembly. Pixie then runs playouts using the setup described in the run specification file. When the playouts are completed it uses the run specification to calculate a final score for the population member, which it records on the genotype.

At the end of a generation we sort the population by score. To create the next generation, we take the highest-scoring member from the previous generation and, if elitism is enabled, add it to the new population (we used elitism for all of the case studies in this paper). We then add two mutations of this genotype. Finally, we add two newly-generated members to improve diversity. This process then repeats with the second highest-scoring member and so on, until the size of the new population reaches a set limit, at which point we repeat the evaluation process. Our case study population sizes were kept around 15-20, for 8-10 generations.

Mutation of a genotype works as follows: first, we randomly select one hook from the genotype's set of hooks. Then we pick a random line in the hook's code. All of the lines before this point are kept unchanged; the selected line and those after it are regenerated. This ensures the scope remains consistent, in case a variable is declared on a line and then referenced later. Evolution proceeds for a fixed number of generations, it does not terminate based on fitness or genetic diversity. This is to ensure the developer has a good estimate for how long the process will take, since they also set the maximum duration of each playout. At the end, all members of the final population are written to files, and labelled according to their score.

Case Studies

Pixie is best understood through worked examples, due to the complexity of the system's internal wiring and the abstract nature of its design. In this section we walk through five case studies in varying levels of detail. Two of the case studies were used during Pixie's development and testing, three were used after the version of the system described here was complete. Videos of mechanics from all five case studies can be found online.[2]

In-Development Case Studies

During its development, Pixie was tested on two open-source Unity games that were not written by the authors. These games were released as part of a tool called Squeezer (Johansen, Pichlmair, and Risi 2021), a research project led by a researcher with a lot of experience as a game developer. Both games were rapidly implemented to demonstrate Squeezer, which made them good examples of 'natural' videogame codebases that were not made for a specific research purpose. One of these was a clone of *Breakout*, an arcade game from 1979, while the other was a casual action game which we refer to as *Dodgeball*.

Breakout Breakout's game logic is mainly contained in component scripts attached to the ball and the pad (or bat) that the player controls. We labelled a range of fields including the acceleration and velocity of the ball, the maximum speed of the ball, the score gained per brick broken, and a method to increase the player's score. We wrote a very simple autoplayer that moves left and right to always track the ball – this is a 'perfect' player, in the sense that it never loses the ball in ordinary circumstances, although it is not the most efficient as it does not try to aim the ball at remaining bricks.

We added three hooks to the game, called when the ball hits either a brick, the screen edge, or the pad. These were all inserted into existing logic within the game. We then ran two experiments with different objective functions: one where the objective was to maximise the player's score, and another where the objective was to minimise it. The motivation here was to discover new potential powerups, items that sometimes fall from broken bricks and change the game rules temporarily if the player collects them.

For score *maximisation*, experiments yielded a range of suggestions. Pixie would commonly suggest code that rewarded the player with extra points for breaking a brick. These solutions might be considered 'reward hacking', as Pixie is directly adjusting its success metric, although they could also be considered similar to a score multiplier. We removed the annotations on the score functions to refocus the search. Pixie also suggested a powerup that increases the size of the ball each time it hits a brick. This is an interesting and unusual suggestion, with an obvious flaw: at some point the size increase is enough to trigger hitting a new brick, which triggers another size change. This chain reaction clears the remainder of the board. This is a good example of how Pixie's suggestions can be interesting and unusual, but also require human tweaking to constrain them or turn them into something balanced for the game.

Pixie also suggested a solution which extended the brick-ball collision logic, reflecting the ball's velocity in the y-axis. This is similar to the logic that is already in place for hitting a brick, effectively reflecting the ball's velocity *twice*, meaning it continues moving in the same direction after hitting a brick. This means it doesn't return to the pad until it hits the ceiling, and so keeps hitting more bricks, resulting in a higher score. Interestingly, a very similar powerup to this exists in most implementations of Breakout, where the ball is able to break bricks and keep moving in the same direction. In computational creativity research, this kind of reinvention of existing concepts is seen as a positive sign for the expressivity of a system (Ritchie 2007).

For score *minimisation*, Pixie's most common result was a solution which made the ball smaller each time it hit a brick or a wall. This solution meets the objective because the smaller ball is less likely to hit bricks, which makes the AI player score lower on average. However, for real human play it would be effective also because it makes the ball harder to see and keep track of. This highlights how the mismatch between automatic evaluation and human playtesting can result in unexpected outcomes or insights into the game's design – as well as results which are entertaining to watch emerge as a user of Pixie.

Dodgeball Dodgeball is an arcade game inspired by an existing independent videogame, in which the player runs around a room dodging balls. Periodically new balls spawn

[2] https://www.possibilityspace.org/pixie

in the middle of the screen and move and bounce around the room. In the full game there are many different kinds of ball with different behaviours (moving at different speeds, tracking the player, and so on). The key components of the game code are the player themselves, the game manager script that spawns balls, and the script governing basic ball movement and behaviour. We attached annotations to the player's movement speed, the logic for spawning balls, and ball properties such as how long they stay still at the beginning before moving, or how fast they move.

This case study leverages Pixie's ability to generate and instantiate objects. We added two new ball types with no special behaviour – a football and a tennis ball. In our test level every fourth ball spawned is a football which has a hook attached to it that is called when it bounces off a wall. The tennis ball is not spawned by default, but Pixie can define its properties and spawn it in its generated code. In addition to the hook within the footballs, we also added a hook which is called once when a level begins, which allows Pixie to add extra logic to spawn objects at the beginning.

The autoplayer for Dodgeball is more complicated than our other case studies. It calculates a repulsive force between the player and every ball on the screen, attenuated by the distance between the two. The player then moves in a direction given by the overall force, with a slight pull towards the center of the level. In addition to this, we added a variant of the autoplayer that adds a delay of a few frames between its decision to act and sending the action. This allows us to simulate slower reaction times and compare players. This level of autoplay optimisation requires more specialist knowledge about AI game playing – we discuss this later in the paper.

In all of our experiments with Dodgeball, we were interested in designing a new kind of ball to change the nature of the challenge for the player. We ran three experiments in total. In the first we set the objective function to maximise the difference in survival times between the two player types; the difference between a 'good' player and a bad one. As expected, Pixie's solutions to this generally make the game straightforwardly harder. Balls which cause other balls to spawn when they hit the wall was common, as was a solution which makes balls bigger when they bounce off walls.

In the second experiment the objective function aimed to *minimise* the difference in performance, which yielded more interesting, albeit extreme results. To minimise the performance difference Pixie can either make the game easier, in which case both agents are likely to reach the time limit alive, or make it so hard that both players die instantly. Pixie found both of these types of solution. In one, it spawns a huge amount of balls at the beginning of the level, completely overwhelming both players. In another solution, it makes balls teleport outside of the level when they hit the wall, removing them from play. This is arguably overfitting, however making the game easier is certainly one way to achieve a narrower skill gap.

Our third experiment involved a modification to the run specification so that it gathered information on the player's position during play. We split the level space into a 5x5 grid, and marked grid segments as the player entered them for the first time. The objective function for this experiment was to maximise the total number of grid segments the player enters. Pixie's solution to this was to create footballs which move at a normal speed, but spawn very slow-moving tennis balls when they hit walls. Over time, this fills the screen with slow-moving obstacles, which made it easy for the player to dodge but required them to move around the level a lot more to find safe paths out of the way of the faster-moving footballs. This solution struck us as innovative and unusual, and is dissimilar to anything in the original game. It also demonstrates how objective functions can drive less straightforward behaviour than simply good or bad performance.

Follow-Up Case Studies

After developing Pixie's core features, we performed a follow-up study using open-source games. We found these projects by searching itch.io and GitHub for open-source games made in Unity. These included games made for game jams and as personal projects. We used three games: *Gravity Golf*[3], a clone of *Arkanoid*[4] (similar to Breakout) and a clone of *Vampire Survivors*[5]. As we are limited on space, we omit the full details of our case studies here, but will summarise some of the important details and observations.

Gravity Golf is a one-button golf game where the player can flip gravity and must guide a golf ball into a hole. Our experiment with this game involved a 'flashtest' rather than an automatic AI player – we wrote a script that performed specific actions at specific times, including clicking a second button (which called a Pixie hook). With no hook, this sequence of actions would not solve the level. We then asked Pixie to evolve a hook, with the objective function scoring how close the ball was to the hole, effectively asking the system to invent a new button the player could press that would let them solve the level in a different way.

Its solutions took advantage of specific features within the level we were testing it on, which could have been adapted into fuller mechanics. In one solution, pressing the second button inflated the ball's size temporarily, which caused it to press against the edge of the level and roll horizontally, which provided enough momentum to move it towards the goal. In another solution, the button accelerates the ball slightly *away* from the goal, but this causes it to drop down onto a sloped tile, giving it enough velocity to roll back and complete the level. This shows that flashtests can be effective for finding new concepts, in the absence of a more complicated autoplayer.

Arkanoid is named after the 1986 NES game of the same name, but is an unaffiliated reimplementation of the game by an independent developer. We included this example to show that two different implementations of the same game can yield different results. We wrote a similar autoplayer to the Breakout example earlier that tracks the ball perfectly, and used the same objective as with Breakout, optimising for score. Pixie invented some similar mechanics, including increasing the size of the ball, as well as some different mechanics which caused the ball to reset near the center of the

[3] https://github.com/HarmonyHoney/Gravity-Golf-GMTK2018

[4] https://github.com/sphynx/arkanoid/

[5] https://github.com/matthiasbroske/VampireSurvivorsClone

screen while retaining its velocity.

The size increase mechanic differed to the one discovered for Breakout. A key difference in the Arkanoid project is the use of Unity Tilemaps for the bricks with a shared collider, whereas the Breakout project uses individual objects which have their own collider. This has the effect of changing how and when a collision event is sent by the game. In normal play both approaches behave identically, however if the ball is big enough to touch two or more bricks at once, the Tilemap collider will not detect subsequent collisions. This meant that growing the ball in Arkanoid still increases score, but if the ball gets too large it will stop detecting collisions altogether. Pixie's solution involved increasing the ball size only when it hits a wall, rather than a brick – something which happens less often, which puts an upper limit on the size of the ball. This gained the benefits of the bigger ball, without making it so large that it softlocked the test.

Vampire Survivors is a casual action-RPG where the player primarily controls movement and upgrade decisions, but not combat abilities. This case study uses an open-source clone of the game's basic features, using Creative Commons art. This is by far the most complex codebase in our case studies, containing hundreds of classes and a complex inheritance structure. We set up an autoplayer similar to *Dodgeball*'s, based on vector forces pushing the player away from obstacles. We created a blank weapon template and attached hooks to three methods: one for when the weapon fires, one for when it creates a projectile, and one for when a projectile hits an enemy. The objective function optimised for a mix of kills, time survived for, and the player's remaining health.

Pixie was able to invent interesting categories of weapon behaviour, but struggled to innovate in certain areas such as patterns of shot or reactive abilities. This is partly due to a lack of library functions for reacting or responding to these events, but also speaks to a need for more complex code generation (such as loops and conditionals). We believe this example reinforces our claim that specific software engineering choices can help support automated systems in writing better code (Cook 2020). Pixie's suggested weapons include a gun whose bullets do low damage but very high knockback, which forces enemies away and creates a lot of space; and a weapon which launches the player backwards, akin to extreme recoil. Pixie struggled to find middleground weapons that killed enemies, partly because the code's software structure made properties such as weapon damage very hard to access.

Initial Assessments of Generated Mechanics

Evaluating automated game design research is difficult, not least because evaluating *human* game design is subjective and something we struggle to do. Evaluation also raises the question of what the goal of automated game design research is. A productivity-focused argument might be that such systems should aim to match or exceed human performance in a task such as mechanic invention. For this project, our personal goals were to develop a system that is easy for non-experts to configure and apply; easy to attach to unseen codebases engineered in different ways; and able to find solutions to design challenges that are *interesting*, unusual or different. Productivity or human competition is not a concern for us, compared to expressiveness or playfulness for designers of all kinds.

We are currently planning a user study to assess how easy developers find Pixie to use, using set tasks and interviews. In this paper, in addition to providing an account of our development process and case studies, we also surveyed a small group of developers to assess their response to some of Pixie's outputs. This is not how Pixie would normally be used, since our participants did not use Pixie, did not make the game it was being applied to, and could not play the outputs. However we were curious whether developers would consider Pixie's outputs usable or interesting even in theory.

We surveyed 17 participants[6], all self-identified as game developers, recruited from online developer communities. We showed them clips of two games being played without modification – *Breakout* and *Dodgeball*. We then showed them four clips of each game, two using human-designed mechanics and two using mechanics designed by Pixie (randomly ordered). We identified the origins of each, and made it clear we were not comparing the performance of AI and human designers. For each mechanic we asked if it was surprising, if it was successful at achieving the stated goal, and if it was clearly understandable. For the AI-generated mechanics we additionally asked if it could be used in a game in its current form, and whether the participant could think of a way to improve it. The mechanics shown were the ones described in the *Case Studies* section above.

Our hypothesis was that developers would find the AI-generated mechanics more unusual than human-designed mechanics, but also less usable without editing and potentially harder to understand. Results can be seen in Figure 1. For *Breakout*, both generated mechanics were rated as more surprising or unusual than the human-authored mechanics, a statistically significant difference (statistical significance thresholds were adjusted using a Bonferroni correction). For *Dodgeball*, the human-authored mechanics were rated as slightly more surprising, however this was not statistically significant. Interestingly, while the generated Breakout mechanics were rated as less clear and successful, the generated Dodgeball mechanics were rated as equally or slightly more clear/successful. This difference is also reflected in the extension questions: the more surprising Breakout mechanics were rated as less usable, compared to the less surprising (but more successful) Dodgeball mechanics.

One factor that may have influenced this is familiarity. All participants stated they had played Breakout or a similar game before, but three participants had not played Dodgeball before, and two were not sure. Dodgeball's mechanics were also aimed at making the game harder, which may be harder to assess than making the game easier, which Breakout's mechanics focused on. These mechanics were picked by the authors from several runs of the system (although all runs resulted in mechanics that solved the problem and that we would have been happy to include). Human-authored mechanics were picked in part based on existing mechanics implemented in the sample games. Our intention here is not

[6]King's College London ethics reg. MRA-24/25-49607

Game, Source (Mechanic)	Surprise	Success	Clarity	Usability	Fixability
HB1: Breakout, Human (Passthrough)	1.71 (0.89)	4.76 (0.55)	4.82 (0.51)	N/A	N/A
HB2: Breakout, Human (Bullets)	2.65 (1.13)	4.71 (0.46)	4.76 (0.73)	N/A	N/A
GB1: Breakout, Generated (Grow)	3.94 (1.06)	4.29 (0.75)	4.71 (0.46)	2.12 (1.18)	4.35 (0.84)
GB2: Breakout, Generated (Reflect)	3.88 (1.08)	4.18 (0.98)	3.18 (1.25)	2.71 (1.13)	3.71 (1.02)
HD1: Dodgeball, Human (Large Tracker)	3.24 (1.0)	4.18 (1.1)	4.35 (1.13)	N/A	N/A
HD2: Dodgeball, Human (Wall Trackers)	3.06 (1.0)	2.65 (0.97)	4.65 (0.76)	N/A	N/A
GD1: Dodgeball, Generated (Scale Up)	2.82 (1.25)	4.65 (0.48)	4.76 (0.42)	3.82 (1.15)	3.82 (0.78)
GD2: Dodgeball, Generated (Spawner)	2.35 (1.03)	4.71 (0.57)	4.12 (1.02)	3.41 (1.29)	3.59 (0.97)

Figure 1: Results from a survey of game developers. All ratings are from 17 participants, on a 1-5 scale. Standard deviation figures are in brackets. Questions 4 and 5 were only asked for generated mechanics.

to draw strong conclusions about Pixie's capability or usability, and we report these results mainly for completeness of this initial paper and to provide another perspective on the system's current state. However these results are encouraging and a positive sign for us – in conjunction with the case studies, we believe it shows Pixie is moving in a good direction as a research project.

Limitations and Future Work

Demands on the User

Our primary goal with Pixie was to explore ways that mechanic generation tools and automated game designers could be attached to unseen game codebases with little specific AI expertise. Some aspects of Pixie succeed at this very well, such as the annotation framework. There are two areas where more work remains to be done: the objective function, and the design of autoplaying test levels.

Objective functions are difficult to write even for experienced AI researchers, especially given the unpredictable ways in which AI can optimise for goals. Besides providing good tutorial resources with Pixie, one point of future work is the inclusion of a `[Goal]` annotation that can be attached to a field (or fields) in the codebase. Adding a `[Goal(MAXIMISE)]` tag to a numerical field, for example, will result in Pixie interpreting it as a simple objective function to increase the value of this field by the end of the playout. This reduces the amount of code being written, and simplifies the concept of an objective function – with the acknowledgement that it also reduces the richness of outcomes possible.

Designing test levels is also a challenge, as some of our richest examples use a simple AI to play the game automatically. This is difficult for many designers, and for many games it is a frontier AI research problem. However, as we have demonstrated in examples such as the Gravity Golf case study, test levels can involve no autoplayer at all, and can instead focus on a 'flashtest' of a specific moment in the game. Clearly communicating this to users and explaining how to build more and less complex test levels will help overcome this. At the same time, we believe that making, observing and changing objective functions is part of the process of working with Pixie, and so we think some friction is acceptable as long as designers feel there is a creative richness and expressivity to working with them. This is another goal for a future hands-on study.

LLM Code Generation

A natural suggestion for next steps might be to replace the code generation module with an LLM, such as GitHub Copilot. Such AI models are now seen as a standard tool for some programmers and are widely used by code synthesis researchers. This is certainly a possible route other researchers may wish to explore; as we state earlier, the exact method of code synthesis is not critical to Pixie's operation. Given the widespread nature of this technology, we thought it useful to spend some space in the paper explaining our decision to use other techniques.

First, we are interested in showing what can be achieved with low-resource, low-cost and no-data approaches. Pixie runs locally on a laptop or computer, using very little resources. Its code generation in particular has almost zero computational overhead – it maintains some simple data structures to keep track of what is in scope, and code is stored as collections of strings and some metadata types. We believe that finding efficient and simple solutions to AI problems is an important contribution research can make, especially as the climate costs of both the games industry and the AI industry soar. It is also worth noting the controversial position LLMs occupy in creative fields currently. New lawsuits are appearing regularly surrounding the way such systems are trained, and anecdotal evidence suggests that some game developers forbid the use of LLM-generated code in projects for both security and IP law reasons (of course some developers are equally using it liberally).

There is also a broader question of whether LLMs are effective for exploratory design tasks such as this. One of the advantages Pixie has is that it has zero preconceptions about how to generate code. It is as happy to create code which causes a ball to grow to the size of the screen, wiping out every brick in sight, as it is to create code which gently accelerates the ball so it hits the bricks faster. Existing studies of LLMs tackling game design tasks suggest they are somewhat conservative in what they generate (Anjum et al. 2024), possibly due to having been trained on generic implementations of popular games.

Finally, there are many questions relating to the role of automation in the creative industries, and any new tool, particularly one which automates new or unusual processes like

Pixie, are under greater scrutiny as a potential threat to jobs, or as tools which disrespect the creative process. We believe Pixie is interesting because it is messy, because it is flawed and, above all, because it requires a human user to work with it. The design of Pixie, from top to bottom, aims to create something the user feels they can customise, direct and control. Integrating an LLM into this process runs counter to this philosophy, if for no other reason than it makes Pixie dependent on a third-party tool which we have no control over. In its current state, Pixie relies on nothing more than Unity itself, and the .NET Standard Framework.

Code Quality

The code generated by Pixie is usually of very poor quality. Redundant assignments, declaring variables that are never used, and unnecessary calculations are just some of the strange things that appear in the code output by the system. We consider this a feature, not a bug: Pixie is not a code generator. It is a system which generates code in order to produce its actual output, which is ideas.

However, especially given our focus on non-expert users, it may be that some of Pixie's code is not clear enough for the user to reimplement themselves, and they may not be able to implement a game idea simply by seeing it in action. Therefore it would be useful to add a subsystem to Pixie in the future that attempts to simplify a code segment. We have also experimented with evolutionary pressure towards simpler code segments with fewer expressions and lines of code, however this has negative impacts on the usability of the system (as it also affects how objective functions must be written). This remains an open area for us to explore.

Expressivity

One limitation caused by the way we define run specifications is that all hooks have access to all annotated fields and methods. This means it is not possible to evolve two hooks at once and restrict their access to different kinds of data. For example, if we were interested in defining an `onSpawn` method and a `onUpdate` method for a creature, we might want to stop the spawn method from accessing any data outside of the creature's own class. This is not currently possible with Pixie, and we suspect that expanding the expressivity of the toolkit would also impact its ease of use. Further studies with developers will help us understand how important this is as a use-case.

There are other edge cases or design patterns that Pixie is not well-suited for. One of our case studies, *Vampire Survivors*, uses a series of object pools to manage enemies, collectibles and projectiles. Pixie will still try to instantiate these objects directly, outside of the established patterns of the project. This is not necessarily a flaw – since our intent is for Pixie to demonstrate ideas which are then reimplemented correctly by the user. However, other games might enforce such design patterns more strictly, making it much harder to Pixie to generate correct code at all, and thus unable to test ideas and return results.

Initially we aimed to represent code using a C# abstract syntax tree, similar to data structures used by compilers. We were unable to find a third-party implementation of this that was also compatible with Unity. Using abstract syntax trees has many benefits, including making it easier to mutate and crossover programs, and so this is an important future work item to expand the expressive capability of the system.

Wider Application

Unity is one of the most popular game development tools in the world, but we believe the fundamental structure and design of Pixie can be applied beyond it to any game engine or codebase. Pixie does depend on some features which not every programming language will have: it must have some way of reading annotations or labels at runtime using reflection, and it must have a way of dynamically compiling or interpreting code at runtime. Even where this is not possible, workarounds may exist – for example, some fields inside the Unity API cannot be annotated because they are closed source, but we were able to provide the user with a way to label them for Pixie using explicit paths.

We are most interested in applying these techniques to Godot and PICO-8 next. Godot is rapidly growing as a popular game engine, and is fully open-source which opens up exciting new possibilities for integrating tools like this into the editor directly. It also supports 'hot reloading' where code is recompiled at runtime, which may make implementation easier. PICO-8 is of interest to us because it is smaller, lower-resource and has a more limited API. This presents new challenges and a different design space from a software engineering standpoint.

Conclusions

In this paper we introduce Pixie, a system for automatically generating short code snippets that solve small design challenges. Our goals were twofold: to explore frameworks for building automated game design tools that are usable by non-experts, and to establish ways for automated game design techniques to be applied to unseen codebases. Using a combination of annotations, metaprogramming, runtime code compilation and automatic playtesting, Pixie allows users to set up small, focused experiments with just a few lines of code, and also allows them to control what parts of their game are included in the design space.

Acknowledgements

Thanks to the reviewers of this paper for their feedback. Thanks to Mads Johansen for his work on Squeezer, and to the developers of all the games we used as a basis for our experiments, including Ivan Veselov who made the Arkanoid prototype. Thanks also to Johor Jara Gonzalez and Niels Poldevaart for great discussions in this space – everyone should read their work (Gonzalez, Cooper, and Guzdial 2023; Poldervaart 2023).

References

Adams, T.; and Adams, Z. 2006. Dwarf Fortress.

Anjum, A.; Li, Y.; Law, N.; Charity, M.; and Togelius, J. 2024. The Ink Splotch Effect: A Case Study on ChatGPT as a Co-Creative Game Designer. In *Proceedings of the*

19th International Conference on the Foundations of Digital Games.

Butler, E.; Siu, K.; and Zook, A. 2017. Program synthesis as a generative method. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*.

Cook, M. 2020. Software Engineering For Automated Game Design. In *Proceedings of the IEEE Conference on Games (CoG)*.

Cook, M.; Colton, S.; Raad, A.; and Gow, J. 2013. Mechanic Miner: Reflection-Driven Game Mechanic Discovery and Level Design. In *Applications of Evolutionary Computation - 16th European Conference, EvoApplications 2013, Vienna, Austria, April 3-5, 2013. Proceedings*.

Cook, M.; and Smith, G. 2015. Formalizing Non-Formalism: Breaking the Rules of Automated Game Design. In *Proceedings of the 10th International Conference on the Foundations of Digital Games*.

Gonzalez, J. J.; Cooper, S.; and Guzdial, M. 2023. Mechanic Maker 2.0: reinforcement learning for evaluating generated rules. In *Proceedings of the Nineteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Guzdial, M.; Liao, N.; Chen, J.; Chen, S.-Y.; Shah, S.; Shah, V.; Reno, J.; Smith, G.; and Riedl, M. O. 2019. Friend, Collaborator, Student, Manager: How Design of an AI-Driven Game Level Editor Affects Creators. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*.

IDV. 2002. SpeedTree.

Johansen, M.; Pichlmair, M.; and Risi, S. 2021. Squeezer - A Mixed-Initiative Tool for Designing Juice Effects. In *Proceedings of the 16th International Conference on the Foundations of Digital Games*.

Keogh, B. 2023. *The Videogame Industry Does Not Exist: Why We Should Think Beyond Commercial Game Production*. The MIT Press.

Lai, G.; Latham, W.; and Leymarie, F. F. 2020. Towards Friendly Mixed Initiative Procedural Content Generation: Three Pillars of Industry. In *Proceedings of the 15th International Conference on the Foundations of Digital Games*.

Lindeijer, T. 2008. Tiled Editor. https://www.mapeditor.org/.

Mateas, M.; and Stern, A. 2003. Façade: An Experiment in Building a Fully-Realized Interactive Drama. In *Game Developers Conference (GDC'03)*.

Mojang. 2009. Minecraft.

Nelson, M. J.; and Mateas, M. 2007. Towards Automated Game Design. In *Proceedings of the Congress of the Italian Association for Artificial Intelligence*.

Poldervaart, N. 2023. Mechanic Miner 2023: Reflection-Driven Game Mechanic Discovery Powered by Go-Explore.

Ritchie, G. 2007. Some Empirical Criteria for Attributing Creativity to a Computer Program. *Minds and Machines*, 17(1): 67–99.

Summerville, A.; Martens, C.; Samuel, B.; Osborn, J.; Wardrip-Fruin, N.; and Mateas, M. 2018. Gemini: bidirectional generation and analysis of games via ASP. In *Proceedings of the Fourteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Treanor, M.; Blackford, B.; Mateas, M.; and Bogost, I. 2012. Game-O-Matic: Generating Videogames That Represent Ideas. In *Proceedings of the The Third Workshop on Procedural Content Generation in Games*.

Zook, A.; and Riedl, M. O. 2014. Automatic Game Design via Mechanic Generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*.