Practical Game Design Tool: State Explorer

Rokas Volkovas
QMUL
London, UK
r.volkovas@qmul.ac.uk

Michael Fairbank
University of Essex
Colchester, UK
m.fairbank@essex.ac.uk

John R. Woodward
QMUL
London, UK
j.woodward@qmul.ac.uk

Simon Lucas
QMUL
London, UK
simon.lucas@qmul.ac.uk

***Abstract*—This paper introduces a computer-game design tool which enables game designers to explore and develop game mechanics for arbitrary game systems. The tool is implemented as a plugin for the Godot game engine. It allows the designer to view an abstraction of a game's states while in active development and to quickly view and explore which states are navigable from which other states. This information is used to rapidly explore, validate and improve the design of the game. The tool is most practical for game systems which are computer-explorable within roughly 2000 states. The tool is demonstrated by presenting how it was used to create a small, yet complete, commercial game.**

I. Introduction

A. Video Game Design is Tedious

At their very core, games are systems built for human interaction. Building these systems is the art form game designers specialize in, driving the evolution of game types. However, we argue, that in terms of the development support, modern video game design is nowhere near as advanced as the secondary skills associated with game development. In other words, the premise of this paper is that designing systems for video games is a tedious process and some of the tedium could be alleviated by exploiting available computational power.

The specific problem identified is that a significant amount of game development time is consumed by play-testing. While some of that time is fundamentally inseparable from game development, we argue that a considerable amount of it is wasted navigating through the game states to inform the next design-iteration decisions. This information could be in the form of ensuring that a state of a game is accessible or not, guaranteeing some values of the game state are consistently reached, and other similar variants.

Examples of wasted time are the playthroughs to make sure that a level in a puzzle game (e.g `Sokoban`) is solvable, playing a level of a Match-3 game (e.g `Candy Crush`) a few times to get the idea of the average number of moves needed to solve it or navigating dialog trees in text-based adventure games (e.g. `A Dark Room`). In all these cases, the information being extracted through the play-testing is technical in that it does not rely on the human experience.

In other words, these types of information extraction can be evaluated by writing specialized tests. However, from the authors' experience, manual play-testing is usually faster than writing and maintaining specialised test code. This is highlighted by the fact that the tests would only be valid for as long as a system is present in the game. Writing tests when design alterations are common is not a good use of time. Can this issue be circumvented?

B. Problem Scope

To approach solving the problem of shortening the iteration cycle of designing and building systems for games, we first narrow down its domain and define the scope of the problem in question.

Our idealized set of features to address the problem of slow video-game design iteration speed were identified. Conceptually, these were the ability to:

- navigate game states manually
- navigate game states automatically
- to identify states of interest
- examine the explored states

Given a tool with these features, a video game designer can delay manually play-testing the game for longer by using the information provided by the tool. Boolean questions, such "is this `Sokoban` level completeable?" or "can the player move through a door without getting the key?" are answered immediately, avoiding unnecessary play-testing.

While there are many more features that can potentially cut down on the design iteration cycle time, these were identified as the crucial foundation ones, upon which further improvements can be constructed.

C. Problem Domain

Having defined the conceptual requirements of the tool, the specific technologies used are still in question. While the individual approach to designing a game may vary drastically from person to person, inevitably, in order to be playable on a computer, all video games have to be programmed on a computer in some way.

From the programming level, any type of further scope reduction leads to sacrifices in ability to express some system descriptions. In order to maintain the generality of being able to program any system (not necessarily a game) and show the viability of the developed tool, a language was selected to implement the tool for. Language generality is a particularly important feature, as iterations in game designs might fluctuate wildly between different types of system behaviour.

The selected language was chosen to be `GDScript`, which is the scripting language used in the `Godot` game engine. Same language was used to implement the tool as a plugin for the engine editor. With this setup, the built plugin can be thought of as another tool in the engine toolbox. All the language features used in this project can be readily converted to other languages, the primary reason for using this specific setup was the authors' familiarity with the environment.

D. *Game vs Mechanics analysis*

Given the nature of the task of speeding up game design, it is important to stress the difference between supporting games and their mechanics. Arguably, non-trivial games are never conceived as a whole. Instead, they are constructed from systems the player can interact with. These systems, or mechanics, are not necessarily games, but are essential components of games. These include the ability to play a card, shoot monsters or steer a car. Many games share mechanics, but differ in the way they are used in combination with other mechanics. As such, they can be viewed as the building blocks of games. This is to say that designing games first requires the design of mechanics; and so a tool, built to support game design, should fundamentally support design of mechanics. To rephrase, the goal of this project was to produce a tool which would be useful even in designing mechanics such as moving on a 2D grid, where there is no winning or losing.

II. Related Work

A. *(Lack of) Game Design Support*

In a 2012 paper [1] Katherine Neil identifies a deeply related issue of lack of tools for game design:

“While nobody would expect an architect, for example, to design a building with Microsoft Word, using natural language to communicate the layout of a building prior to construction, game designers are required to do something somewhat analogous to this.“

She argues that in order to overcome the hurdle and advance the research in the field of game design support, real world data of how game design is conducted in practice is necessary. In the eight years since the analysis, little has changed. She goes into more depth in the topic in [2].

Notably, game design suffers from being not only a relatively new art form, but also a tremendously time-consuming one. Specifically, while the introduced tool uses the design of a single game as a data point to show how game design is done by some portion of smaller developers, it nowhere near represents the entire industry. Furthermore, the design and development of the game, by a single person, took roughly three months of full-time work with some experience in game development. Design taking roughly 70% of the work, about 30% of it mechanical, other 40% being the visual design, which also contributed to the mechanical design decisions. In other words, getting the smallest and cheapest possible useful data point in game design consumes three months of time. This makes the lack of tools supporting game design more understandable, but no less bearable.

Another interesting point to notice is that `Microsoft Excel`, the popular spreadsheet software, is a common skill new applicants for game design positions are expected to have. The following is a brief list of larger companies listing the skill as required or at least desired for a game designer position live at the time of writing: Ninja Theory, Creative Assembly, SEGA Europe Limited, ZeniMax Media Inc, Machine Zone. The presence of such a general tool at the core list of skills sought after reinforces the perceived lack of tools supporting the process of (video) game design.

B. *Machinations*

While there is no tool which would would ubiquitously used or at least known by game designers, there are definitely practical and research attempts at establishing one and `Machinations` [3] is arguably the most prominent example. Figure 1 shows the basic visuals of the program - a dedicated design application with a UML diagram-like interface.

The primary goal the program attempts to achieve is the separation of the system development from games as much, if not more, as you would with 3D modeling software. It does this through the creation of a whole ecosystem for mechanics development. It can be argued that this is both its greatest advantage and disadvantage.

Through isolation it can make assumptions, allowing quick access to features not easily integrate-able in traditional programming. On the other hand, the relative iteration speed comes at the cost of a large up-front knowledge load for the person learning the tool and a lack of generality: constructing a game in machinations can be seen as building a model of the game, not the actual game itself. This is the most significant area where the introduced tool differs. We argue that a self-contained development environment is a milestone that is to be reached eventually, but at the same time feel that reaching that milestone must step through a series of cornerstones, bridging the available technology with that of self-containment.

Much like `Blender` (3D modelling software) exports to many different standardized 3D model representations that game engines can take and interpret, a tool created for the purpose of building mechanics must integrate with the most prominent tools used already; all without a human involved, as is the case currently. To achieve this, a programming language library like interface with the developed prototype should provide a lesser barrier of entry for the designer already programming prototypes and introduce the concepts necessary to make the transition to standalone tools.

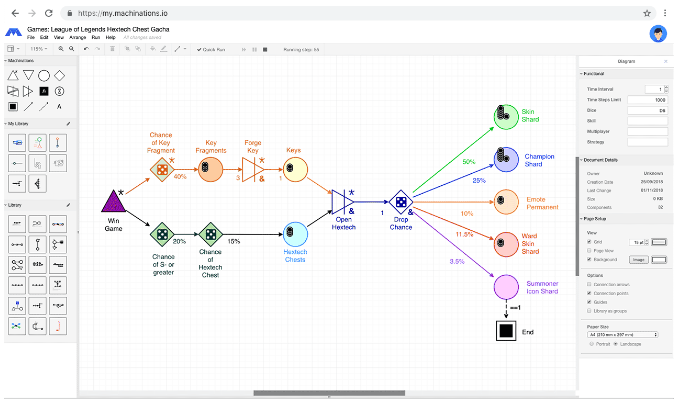


Figure 1: Interface of Machinations, the tool attempting to solve the most similar problem to the tool introduced. Taken from machinations.io [Accessed: 2020-03-16]

C. *Other Work*

Other work done in the field of improving game design iteration speed largely focuses efforts on automated solutions to common problems through procedural generation (PCG) or automated play-testing, with algorithms employing heuristics to make (sometimes crucial) decisions. While not directly related, the following research could potentially become a natural extension or, at the very least, conceptually influence the direction taken with the tool introduced in this document.

In [4], the concept of merging states into hyper-states to simplify game state graphs is presented. Interestingly, the information extracted relies on the ability to have the game states easily explorable (and in this case, fully explorable), but there is no environment to support this type of analysis for general game design. The introduced tool sets this type of environment up, allowing more state exploration based information extraction, including learning curve extraction [5].

Additionally, most of the modern automated game exploration work in PCG and general game AI play-testing, which are attempting to aid the problem of improving design hurdles through different means, seems to stem from the ideas put forward in [6]. These ideas build on the desire for tools to work along side the designer, if not to outright replace her. For example, in AI play-testing work such as [7] the prevailing fundamental assumption is that the agents playing the (usually nearly complete games, at which point no significant changes are expected) games can roughly simulate human play. Much the same way, procedural generation, even in ambitious work such as [8], takes away fundamental control from the designer, which we attempt to explicitly shy away from and build on the core idea of building tools that can speed up certain designer processes instead of guiding them.

This is not to say that work done in these areas is not helpful or cannot be transferred to suit our outlined goals. On the contrary, the algorithms developed for utilizing general game playing agents [9] or generating levels [10] may well be something a designer would love to see. The largest hurdle stopping her from doing so, we argue, is the complete shift in workflow needed to make use of these advances.

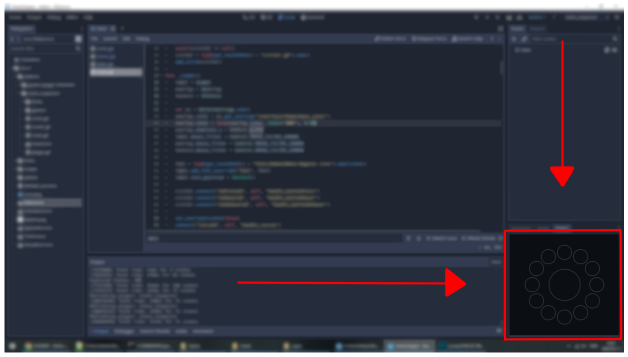


Figure 2: Tool Context: the introduced State Explorer iteration is implemented as a plugin in the `Godot` game engine. Image above points to the SE view integrated alongside the basic engine features of the text editor, file explorer and others.

III. State Explorer - SE

The tool introduced in this paper was given the name of `State Explorer` and abbreviated to `SE` or `tool` from this point onwards. To minimize confusion, this section first describes the functionality of the tool, followed by the description of its current, rather obtuse and less important, visual interface, along with game system exploration examples.

A. *SE features*

Fundamentally, SE is built around three primary features:

- inspection of user code
- visualization of user code states
- user code state navigation

Code inspection and visualization features are commonplace in code debugging. Adding state navigation can hopefully translate their utility to game design. Inspection allows constructing the system state and navigating it in order to find states of interest, while visualization displays those states back to the user. Given these features, the tool visual interface has to display the current state alongside the list the states of interest, reachable from the current state. Symbolic representation of the tool is presented in Figure 3.

B. *SE is a plugin*

The first step to understanding how the tool integrates into the designer's workflow is internalizing that SE is a plugin – an optional extra feature – in a development environment. At a high level, it serves the same purpose of a file explorer: it allows the user to find relevant information quickly, but is not strictly required for the purpose of developing a game. Figure 2 shows SE as one of the panels in the `Godot` IDE. For further emphasis, State Explorer is `not` a tool for designing systems; it `is` a tool for facilitating the design of (game) systems.

As an analogy, SE is like a voltmeter used to quickly determine the voltage between two points on an electrical circuit instead of solving a complicated series of equations every time the circuit is changed. Depending on the type of the project, a voltmeter may vary in how useful it is. Similarly, SE provides some information, otherwise not easily available, but the relevance of that information varies depending on the system being designed.

```
                          # # # # # # #
tool output               # P . . . . #
>starting state           # . . . . . #
 relevant state  1        # . . B . . #
 relevant state  2        # . . . . . #
 relevant state  3        # . . . . . #
                          # # # # # # #
```

Figure 3: Symbolic representation of the introduced tool, State Explorer, visual interface. The tool provides the user with the list of states (left) and the state visualization (right), implemented by the user.

C. SE visualizes states

To deliver useful information quickly, SE uses a designer implemented state visualization. The visualization can take two forms:

- string of text
- pixel data

Text visualization is common in early prototypes, when *rendering* to the command prompt / terminal window and later iterations tend to have visuals drawn to screen. In both cases, no extra work is needed apart from passing the already existing information to the tool.

In other words, when developing a game system, the ability to represent it visually will inevitably be implemented, this representation is what can also be used by the tool to show states. Furthermore, since the tool makes no assumptions on what is being drawn, any type of information, as longs as it is presented as a string or an image, can be selected to be shown on drawing a state.

D. SE representation structure

Figure 4 shows the visual interface of the tool, which mirrors the symbolic representation in Figure 3. This is what is seen in the engine panel dedicated to the tool. The interface is made of three main parts, outlined and labeled as `A B C`:

- A (large circle in the middle) - the active state
- B (small circles surrounding A) - the next relevant states
- C (area overlaying A and B) - game visualization location

That is, the whole center area `C` is used for drawing parts `A`, `B` *and* the game visualization as a transparent overlay (disabled here). The circles represent game states. Large, always present, center circle `A` represents the currently active state, while the surrounding circles `C` represent the next relevant states reachable from the currently active state.

The circles are interactable. When hovered over using the mouse, the tool draws a semi transparent visualization returned by a method implemented by the designer overlaid on top of the circles. When a circle is clicked, the tool makes the associated state the active state – it becomes the large center circle – and the next relevant states are regenerated and the interaction cycle resets.

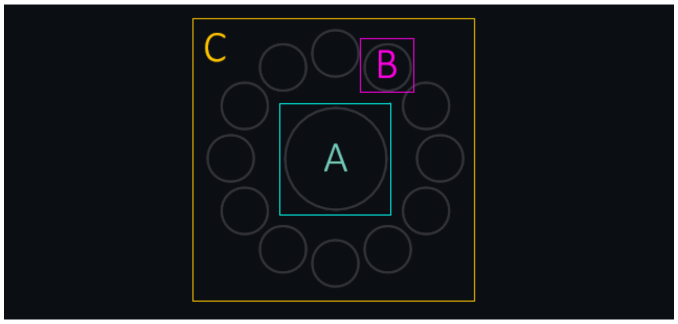


Figure 4: State Explorer visualization: the image above shows the current (at the time of writing) iteration of the tool interface without a game visualization overlaying it.

E. Next Relevant States

The central value of SE lies in its ability to navigate through the game states and find the `next relevant` states. Conceptually, the tool explores all possible game scenarios exhaustively and asks the designer whether or not the encountered state transitions should be shown in the visual interface, whenever the answer is positive, the state is marked as a next relevant state and is no longer explored. When the exploration is complete, the active state with all the discovered NR states are presented in the 4 format the designer can interact with manually.

An important distinction to highlight is the difference between next relevant states and the next available states. For example, in `Sokoban`, the next *available* states are always the possible moves the player can make. However, the next `relevant` states can be the states where the player has just pushed a box or whenever the player moves on to a specific tile or when a box is pushed into a location where it can no longer be moved etc. The choice of what the relevant states are is left entirely to the designer.

The next relevant states are the cornerstone of what makes the tool useful. They give the designer full exploratory control and probing of information that is relevant at the time, skipping the irrelevant states. For example, if it is crucial that the player can push a box into a certain location (still `Sokoban`), every time the level is changed, the designer would have to step through the states (mentally or through playing) to ensure that is the case, whereas the tool would provide the information almost instantly, assuming the information is within reach. ‘

F. State Explorer State Exploration

In order to implement the search for next relevant states, the tool reads the designer code and constructs a starting state by instantiating the root object and repeated copying and modifying the state. To allow for this search, and in turn use the tool, the user must implement two methods:

1) `transition_isRelevant(from, to)`
   returns a boolean indicating whether or not the transition from one state `from` to another `to` should be shown
2) `get_actions()`
   returns a list of actions the tool should try applying to the state the method is called upon

Listing 1 shows the high level algorithm used, which is a slight modification of the Breadth-First search algorithm. The main changes are that search is continued beyond finding a single relevant state and that relevant states are not expanded.

```
func explore_state(state):
  for action in state.get_actions():
    var nstate = state.apply_action(action)
    if nstate.was_seen(): continue
    if transition_isRelevant(state, nstate):
      relevantStates.append(nstate)
    else:
      toExplore.append(nstate)
```

Listing 1: Pseudocode logic of SE state exploration

G. *SE user guidance*

The tool in its default mode of operation is equivalent to a brute-force explorer. Exploring all potential states can be costly and completely unnecessary if the designer knows where to look for the information. There are two input sources of guidance the designer can provide to the tool, avoiding unnecessary exploration:

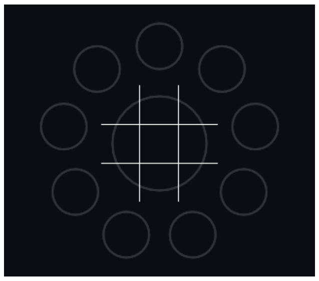
1) restricting the list of actions to the ones of interest
2) restricting the state variables to the ones of interest

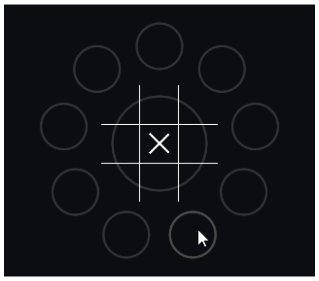
Action restriction allows the designer to explore only a specific set of actions, by excluding others from the list when `get_actions()` is called, forcing the tool to skip considering those states. This can be useful in cases where the player has multiple paths in the game but only one of them is being actively developed.

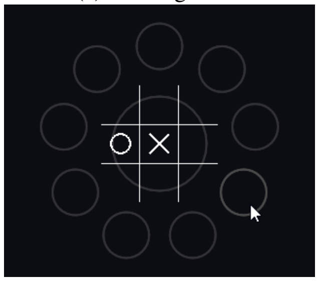
Similarly, restricting what state variables are being tracked for changes allows the tool to ignore changes which have nothing to do with what is being explored. For example, the movement of an enemy on the other side of the map is often irrelevant to checking how high the player can jump.

IV. Sample Mechanics

To test the waters of the tool's viability and prior to committing to a serious project, a variety of well established game mechanics were implemented. The following examples showcase quickly showcased the benefits and limitations of `State Explorer`. The goal in these examples was to develop the entire game or one of its primary systems and see whether or not the tool supported their development. Note the use of the word *development* instead of *design* since the games are already designed, but potential opportunity for change was paid attention to as well.


(a) Starting State


(b) Future State 1


(c) Future State 2

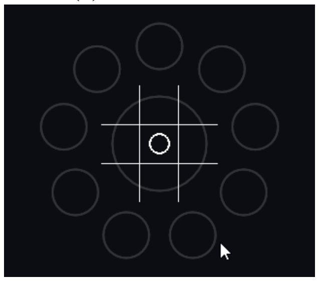

(d) Future State 3

Figure 5: The images above show the tool in use exploring the states of the in-development `Tic Tac Toe` imitation.

A. *Mechanics of `Tic Tac Toe`*

. The first system implemented was that of `Tic Tac Toe`. The logic of the game is as follows:

- there is a 3x3 board
- 2 players take turns in placing distinct symbols on it
- player with 3 aligned symbols wins

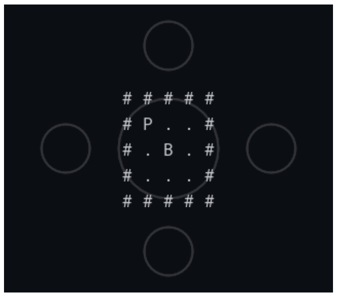
To implement it, an 2D array variable for the board, holding the values of each cell was created. This was followed by implementing the logic for drawing the symbols on the screen. Finally, 9 keys representing the different placement positions were mapped to the actions of placing the player symbol. At this point the game could be play-tested by manually running the game and placing the symbols by pressing the appropriate keys, which appeared to work correctly.

Connecting the implemented logic to SE was trivial as all it needed was one function returning the 9 actions and another implementing the state drawing, which already existed. Figure 5 shows the tool being used with (a) showing the default game state, (b) one of the relevant ones from (a) and (c)(d) showing relevant states from state (b). Interestingly, (d) shows that it is possible to draw a circle on a cross, highlighting a, in this case unwanted, design implementation flaw, which was not discovered on manual play-testing.

B. *Mechanics of `Sokoban`*

The second system implemented was the movement and pushing mechanics of `Sokoban`, which feature a player navigating a 2D grid of open tiles using the adjacent directions for movement, not being able to move into walls and pushing boxes when moving into them if there is an open tile behind them. Similarly to the previous system, it was implemented using a 2D array with values, this time using only 4 keys as necessary for the player movement.

This time the game was initially represented using text and playable in the terminal window. This was also easily connected to the tool by passing it the string of the game state as the visuals. Figure 6 shows the resulting tool interface, with the active state shown on the left and the next relevant state on the right hand side.


```
# # # # #
# P . . #
# . B . #
# . . . #
# # # # #
```
(a) Current Game State

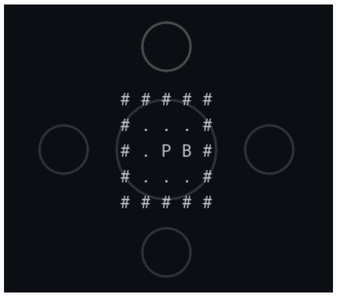

```
# # # # #
# . . . #
# . P B #
# . . . #
# # # # #
```
(b) Next Relevant Game State

Figure 6: Images above show the developed tool in action. The tool shows the starting state and the relevant future state of `Sokoban` mechanics, where the symbols `P.B#` represent the player, floor, box and wall, respectively.

The left image shows the player standing the corner of a 3x3 walled room. The next relevant state, in this example, was chosen to be the state when the the box is moved. The tool correctly detected the existence of 4 such possibilities (4 small circles) and one of them is shown in the right hand side image, indicating correct system implementation.

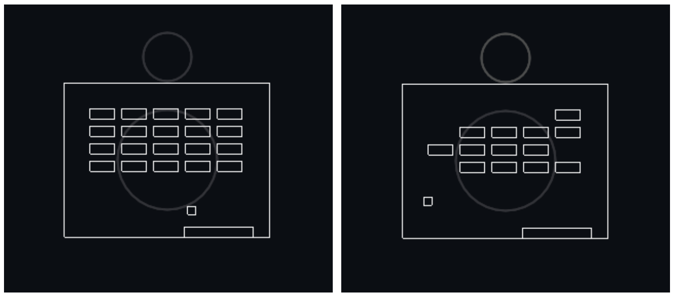
Taking things further, the graphics were replaced with colored squares to indicate the different tiles. The result could the be also passed on to the tool. This system implementation showed the usefulness of the tool through its undo feature, allowing to take turns back, without the logic for it being implemented in the game itself, speeding up the feature testing time, skipping the need to restart the game.

C. *Mechanics of* `Breakout`

The final smaller system implemented using the tool was one imitating `Breakout`. In contrast to the previous systems, this one is meant to be played in real-time, meaning there are many more states to step through. The game works by the player controlling a paddle, moving it left and right to bounce a ball into a wall of bricks, which get removed on hit. Regardless of the game genre being vastly different, it was integrated with the tool just as easily as the previous examples. Figure 7 shows the implemented visuals in the tool panel.

In contrast with the previous examples, here the guidance features of the tool became crucial to gain useful information quickly. For example, in the figure, the next relevant state, was chosen to be the state when the ball returns below a certain screen height position. This forced the tool to show the bricks removed by the ball after it bounces around the outer screen edge or, which allowed measuring the possible angle range at which the ball performs the trick with ease.

Furthermore, this was only possible to do quickly (within roughly 2000 states and under 2 seconds upon saving the code) when only the state of the ball was tracked, explicitly ignoring any state changes with the ball in the same position, but different paddle position.



(a) Current Game State (b) Next Relevant Game State

Figure 7: Images above show the developed tool used in development of an imitation of `Breakout`. Left image shows the starting state, while the right one shows the only next relevant one, skipping irrelevant ball movement states.

V. Eternower

In order to sample a data point of how useful the tool is in a more realistic design environment, it was used in the development of a small, but an original and complete commercial game[1], called `Eternower`. The game is a grid-based tower-defense game and this section outlines how it works and why it was chosen as a useful type of game to evaluate the potential of the tool for practical game design. A screenshot of the game is presented in Figure 8.

A. *Tower Defense*

While the game was set out to be an original game, it was chosen to be built in the area of tower-defense (TD) games. TD games typically work as follows:

- There's is path with a source and a destination that can be traveled on
- Every level, a number of enemies appear on the source of the path and start moving the the destination As soon as they reach the destination, they reduce player's health
- Player loses when her health reaches 0
- Health loss is prevented by strategically placing towers, which shoot the enemies
- Towers cost money, money is received by killing enemies

What makes a TD a good genre for exploring the nuances of sE is that it relies heavily on the designer striking a desired balance between a large number of state variables, which at the very minimum include: enemy health, speed, reward and number, player health, available gold, tower selection and tower positioning. Tweaking any of the values slightly typically changes the gameplay drastically, changing the viability of the established strategies.

A common way of implementing a TD game is to have a set constant path and tower locations where the gameplay is to select the appropriate towers and choose how to upgrade them to keep up with strengthening enemies. This provides the game with predictability and gives more control to the designer, which makes it less useful for tool evaluation. To counter this drawback, mazing and upgrade paths, less common features in the genre were added.

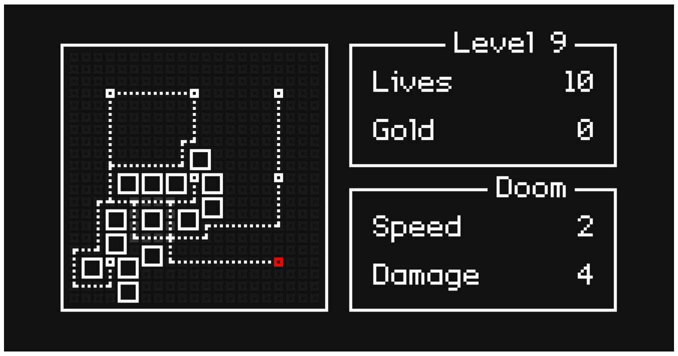


Figure 8: A gameplay screen of Eternower, a commercial game developed to explore the potential of State Explorer

[1]Steam: https://store.steampowered.com/app/1031790/Eternower/

B. Mazing

`Mazing` in TD games is the ability to place towers on the same path the enemies walk on, forcing them to take alternative paths. This in turn results in encouraging players to maximize the amount of time enemies spend walking along the path, getting damaged by towers by building mazes out of said towers. Mazing adds a significant part of complexity to the game, which makes the game much more difficult to design due to the number of possible maze configurations.

Consider Figure 9. The numbers on the left-hand side shows the default path enemies take: they spawn at node labeled 1 and move through nodes 2 - 7. When 7 is reached, the player lives count is reduced. Both images side by side illustrate how different the game board state can look depending on the fort placement. Anticipating all the possible variations becomes a nearly impossible task for the designer, forcing her to rely on intuition, potentially missing some game breaking (compared to what was intended) plays. These mazing properties make it a good candidate for testing the game design tool viability.

C. Upgrade Paths

Another feature to increase the difficulty in designing the game was the introduction of upgrade paths. Upgrade paths are a system that allows the player to choose how the game is modified over the course of a single play-through. It works as follows:

1) Player completes a level
2) Two game-modifying options are presented
3) A new level can only be started when an option is selected

The reason this is a significant design complication is that each one of the options drastically change the way the game has to be played in order to avoid losing. Options include modifications such as: changing the health or speed of all future enemies, allowing to build a new fort for free, giving the player more resources or changing the way enemies move.

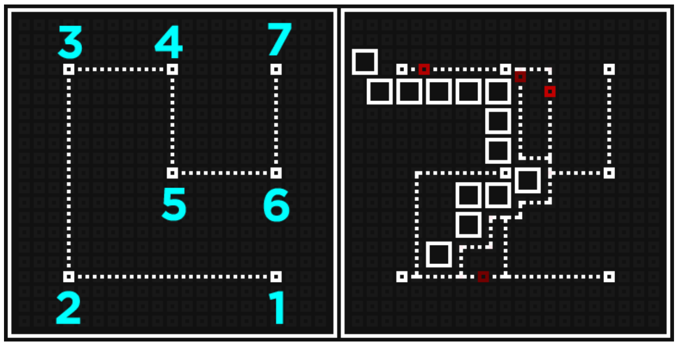


Figure 9: Eternower mazing: left figure shows the constant starting state of the game, while the right one shows one of the potential player-constructed mazes (using large white-outlined squares as towers) and their impact on the path anti-nodes move through. The maze shaping is the primary source of complexity (and potentially attraction) in the game.

All of these features together make designing a game difficult when a certain balance is desired, making them ideal for introduced tool evaluation.

VI. Tool utilization

To reduce the scope of evaluation, the tool was used when the game idea was conceptually established. That is, the genre was preset as were the basic pillars of mazing and upgrade paths. Any further decisions were considered to potentially be aided in exploration with the tool. There were three main use-cases identified for the tool while developing the game: fort placement location scouting, potential level data modification through fortunes (upgrade paths) exploration and level resolution.

A. Upgrade paths

One of the core game systems is the ability to modify the future levels via options presented to the player at the end of each level. For example, at the end of a level the player can choose to either increase the health of all future enemies by 10 or increase their total number per level by 1. Many of the choices directly impact the stats of later levels, making predicting the possible stats of say level 10 difficult as there are 2^{10} possible option paths to reach it.

Using the tool, this information became readily accessible and allowed making decisions about balancing that would have otherwise taken a while before they would have been evaluated due to the time it took to get to level 10. Specifically, using SE, with a single line of code, the tool reports what the different health values are possible. Knowing the range allows immediately throwing away or at least reconsider fortune paths which lead to impossibly high or low levels of health.

B. Path length

The second most clear use for the tool was the ability to examine the available tower arrangements quickly. Identifying the possible placements and how they impact the game is crucial to ensuring a desired level of balancing is achieved. Given a board setup the tool could be used to determine what are the possible modifications to the current paths and what the modifications do to the path length.

C. Level Resolution

The most crucial and time saving use case of the tool was to resolve levels without playing them. Path length can be no shorter than 60 steps. Each steps takes 0.3s of animation time at default gameplay speed, resulting in waiting 18 seconds every time a level configuration is to be tested. Using the tool this can be skipped, which means that if the same amount of time is spent for testing the levels, more information is extracted due to the faster information gathering speed.

Using the tool this way also changed the mindset of approaching balancing. This way much less is left to estimation and *what feels right*, because the information can be so readily gathered, leaving the focus to the more subjective experiences of animations and color palettes.

D. SE integration

It is important to emphasise that the tool is not a magical machine automatically extracting useful information, helping the designer. If used incorrectly, the tool quickly becomes more effort to use than than to extract the wanted information from the game manually.

Specifically, when integrating the tool with Eternower, the game was already relatively far along into development in terms of the final features it had. The part left unfinished was the design and tuning of the systems, such as defining what towers, enemies, levels and each of their abilities are available when. It was found that integrating the tool with a game in this stage of development was not a smooth and easy task.

The issues encountered were similar to those of adding a new library of handling a specific task, that is done implicitly across all the existing program systems. The most largest issue being the lack of system isolation. That is, most of the data that is relevant to the tuning is spread across a number of files, which also handle logic for visual representation of the system. Including animation data increases the amount of data the tool has to keep track of at least tenfold.

In order to make use of the tool, the mechanics had to first be separated as much as possible, which took a considerable amount of time. However, the type of refactoring needed was not unlike ones occurring when system focus changes in other software. For example, many games are built with only the English language available. It is not uncommon for developers to hard-code the strings used for all text in the game in the files that need them. This is the fastest way of building software, but only until the developer finds herself wanting to localize the game. Now, all the hard-coded string constants have to be replaced by hooks, getting the appropriate string from an isolated object along with writing new logic to replace those strings when the language is changed.

This discovery unearths the topic of the game logic isolation. The questions of "where does the game logic stop and the aesthetics begin?", "should there be a custom format for defining mechanics (much like there are a number defining 3D models or audio files)?" and many others appear. While interesting to explore, the topic is beyond the scope of this document and the purpose of this subsection is to underline the need for the isolation of the mechanics to streamline their tuning and, as a side-effect, allow for easy SE integration.

In the case of Eternower, the tile fort node, anti node and path positions used in the logic were directly the positions used by the sprite objects that are drawn on screen. This highlights that while data is the same, it does not mean that is should be shared, in this case it is a (potentially foreseeable) coincidence. To separate the logic out, all data related to fort nodes dealing damage to anti nodes and player stats was first moved to separate objects, established connections without reference to any other game system before finally integrating it back to the complete game. The final interface used with the visuals could also be reused by SE.

VII. Conclusion and Future Work

In this paper, a practical tool for game design, called State Explorer (SE), was introduced. The problem of slow iteration game system design iteration times, in contrast to other art forms associated with game development, was identified. The problem was approached by devising a tool to automatically explore the design from a given state and return information about potential states as requested by the designer.

The tool was first shown to provide information of potential interest to a designer in select few distinct game systems prominent in older video games. The tool was then explored in more depth through its exploitation in the development of a small feature-complete, commercial game `Eternower`. While in general the tool was found to be irreplaceable in the process of making both small and significant design decisions, it was found that the tool excelled more in verification problems rather than exploration.

To make the tool even more useful in game design iteration, the future work on it is planned to improve the robustness of the system along with the interface improvements. Additionally, it is worth exploring the possibility of integrating more sophisticated algorithms into the tool or even expose the hooks necessary to support custom ones for more flexibility. Improving the tool in these areas should eventually allow it to be easily adapted into larger scale games, requiring less in depth knowledge of its inner workings.

Acknowledgment

This work was funded by the EPSRC CDT in Intelligent Games and Game Intelligence (IGGI) EP/L015846/1.

References

[1] Katharine Neil. Game design tools: Time to evaluate. In *Proceedings of the DiGRA Nordic Conference*, 2012.

[2] Katherine Neil. How we design games now and why. https://www.gamasutra.com/blogs/KatharineNeil/20161214/287515/How_we_design_games_now_and_why.php. Accessed: 2020-03-16.

[3] Joris Dormans. Machinations: Elemental feedback structures for game design. In *Proceedings of the GAMEON-NA Conference*, pages 33–40, 2009.

[4] Michael Cook and Azalea Raad. Hyperstate space graphs for automated game analysis. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2019.

[5] Rokas Volkovas, Michael Fairbank, John Robert Woodward, and Simon M. Lucas. Extracting learning curves from puzzle games. *2019 11th Computer Science and Electronic Engineering (CEEC)*, pages 150–155, 2019.

[6] Julian Togelius and Jurgen Schmidhuber. An experiment in automatic game design. In *Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On*, pages 111–118. IEEE, 2008.

[7] Sinan Ariyurek, Aysu Betin-Can, and Elif Surer. Automated video game testing using synthetic and human-like agents. *IEEE Transactions on Games*, 2019.

[8] Cameron Browne and Frederic Maire. Evolutionary game design. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(1):1–16, 2010.

[9] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Simon M Lucas, and Tom Schaul. General video game ai: Competition, challenges and opportunities. In *Thirtieth AAAI Conference on Artificial Intelligence*, pages 4335–4337, 2016.

[10] Ahmed Khalifa, Diego Perez-Liebana, Simon M Lucas, and Julian Togelius. General video game level generation. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pages 253–259. ACM, 2016.