A Game Genre Agnostic Framework For Game-Design

Jonathan Tremblay
School of Computer Science
McGill University
Montréal, Québec, Canada
jtremblay@cs.mcgill.ca

Clark Verbrugge
School of Computer Science
McGill University
Montréal, Québec, Canada
clump@cs.mcgill.ca

***Abstract*—Designing games can involve long waiting periods between design phases in order to get feedback from game testers playing the game. In this work we propose a genre agnostic framework for design, using search techniques from artificial intelligence to generate player traces. We evaluate our framework by presenting three very different games that are sent to the tool and which give relevant design information. This approach greatly accelerates the design process.**

Introduction

Designing video game is a complex, iterative process involving a loop between game developers and testers. With the data produced by game testers, developers alter their design in order to match a desired player-game interaction. The use of human testers requires significant time and effort, making the overally process relatively slow and expensive.

This work proposes a genre agnostic framework for using Artificial Intelligence (AI) as part of the game design. The purpose of such a tool is to accelerate the iterative design, by letting developers send their game and levels to an independent service that uses advanced AI techniques to play the game and collect data. This data is returned and presented to the developers orders giving near interactive feedback on game design.

In theory the presented framework only needs the user to provide a game representation and rules. In this work we present three different game implementations using the framework: a platformer game implemented in Unity 3D [1], a stealth game [2], where the player has to avoid enemy fields of view to reach particular positions, and an attrition game, where two teams fight taking turns. Each game genre is accompanied by one example of using the AI-design tool. It is important to mention that some results discussed in this paper have been disseminated in other papers [1], [2]. Our work thus includes the following contributions:

- The general design of our genre agnostic framework which enables AI as a game design tool.
- Three different implementations, showing the tool is effective in a variety of genres.
- A discussion of how the design enables game developers to verify and visualize whether their game design meets their intent.

Related Work

This work incorporates game solvers and AI techniques into game design tools such as Unity 3D. Game solvers have a long history of application in game analysis, and have clear application in allowing designers to explore various aspects of level design. Designers may ask a variety of fundamental questions about their designs, such as whether and how something might be possible in the game [3]. The need for game-independent frameworks which can be reused for different games genres has also been recognized [4], and is the main concern we try to address in this work.

Our work can be seen in the context of several research efforts which aim at building design tools that incorporate AI techniques into the creation process for specific genres. Jaffe *et al.,* for instance, explore the concept of *fairness*/game-balance in games based on win-rate [5]. Using different search algorithms, such as greedy search, they showed how strategies in a card game could change with different card values, and so give designers important balancing information. Shaker *et al.* presented an approach for solving continuous-time game constraints within the game *Cut the Rope*. They discretized time, finding the next game state by using a physics simulator to update the world [6]. From this update, the available actions are then explored using depth-first search. This allowed them to present a path in a tree of game actions that solves the level. An interesting, high-level approach was given by Pizzi *et al.,* abstracting the level structure and gameplay of the popular video game *Hitman: Blood Money* into discretized events that could be assembled into storyboards [7].

Framework Design

In this section we present a general architecture for game analysis based on an AI tester, which we will then use in three different implementations. We also describe the game state-space representation, and the implementation of our random-oriented solver.

Our framework is intended to be relatively agnostic to the game genre. In this sense it works as a service that game developers and designers could use to test their games and receive meaningful data from the AI game tester as a proxy for real player data showing possible in-game solution. To do

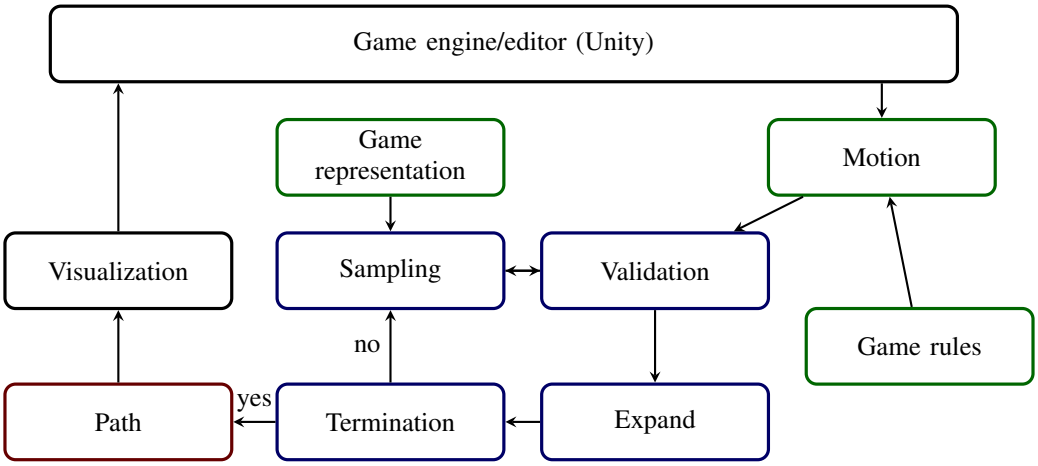

Fig. 1. Structure of our design. The black cells represent the parts provided by the game context, green cells are the parts provided by the user (game developer). Our search process is composed of the blue cells, which generates a visualizable path (red).

this a game designer would need to provide the framework with a formal game representation, and model of the game rules.

Figure 1 presents the general architecture. The design is highly inspired by the Randomly Rapidly exploring Tree search (RRT) algorithm [8], although it accommodates other search algorithms, and is integrated into a level analysis tool. The user needs to provide the game representation, including a level with start and goal positions to initialize and terminate the process, and a state budget for bounding the search costs. A motion model, informed by game rules that defines allowable behaviours and measures state difference must also be provided.

The process starts by sampling a state from the game representation; for a simple 2D movement game (*e.g.* pacman), this is any point position within a level. The validation step checks if it is possible to reach the sampled state from already explored states, using a motion planner or motion verification to check if the sampled point is indeed reachable. This motion system can be provided by the game rules, or directly derived from the game engine physics. If the sampled node is non-reachable, the process returns to the sample step and repeats. If the sampled state is reachable, it is added to a data structure to keep track of what is already explored in the expand step. At this point the process verifies if a terminal state was reached, either by reaching a goal position or exhausting the state budget. This process encompasses multiple search algorithms using a given state space definition, and can be called multiple times to generate a collection of solutions. The framework also provides different visualization, such as movement heat maps, which can be incorporated into the game engine.

State Space - To simulate game play, we are interested in finding paths from a starting position to the goal within the abstract state space of the game. Notationally, the state space is denoted Σ, with $\sigma \in \Sigma$ describing a specific point. For a path, we are interested in a set of ordered points, p, that links σ_{init} to $\Sigma_{goal} \subseteq \Sigma$. Goal states are in general subsets, as they often describe a geometric region rather than a single point, or otherwise allow for a range of values within different state space dimensions (health greater than 0, a time-range, any amount of ammunition, *etc.*),

For a given state, the value of any particular dimension is found and set by simple accessor functions: *e.g.* $x(\sigma)$ returns the x position of the state, and $x(\sigma) \leftarrow x'$ updates its value. Σ_{free} defines the state space where the player can move freely without intersecting with the obstacles, hence any $\sigma \in \Sigma_{free}$ is a valid position for any player, in the absence of other state constraints. To modify state we assume a set of actions, A, where each $a \in A$ is an action that can be applied to a state to generate a new state: $\sigma' \leftarrow f(a, \sigma)$. Some actions might be given a time component, with $f(a, t, \sigma)$ describing an action a applied over a period t.

Our design relies on a distance measure between two states, given by $d(\sigma, \sigma')$. This can be just Euclidean distance in the geometric dimensions, although in general abstract dimensions will be important as well, especially for more abstract game genres.

Solver - Using this general presentation of state space and framework setup, we can play-test a level of a video games by searching for paths. We mainly use the RRT algorithm, as a well known pathing algorithm in the robotics domain that easily incorporates high-dimensional state spaces. Other search algorithms can also be used. In the case of A*, for example, the sample step is determined by a heap data structure ordered by the heuristic nature of A*. We have also experimented with Monte Carlo Tree Search (MCTS), where the node selection rules are defined in the sample process [1].

Experiments & Results

In this section we present three different game domains which use our framework for play testing: platformer, stealth games, and attrition games. For each game, we first describe the game representation followed by some interesting results.

Platformer - Our representation is aimed at the classical platformer game genre, where the game level is fundamentally

a 2D Euclidean space, with time added as a third dimension. Time is important as it gives us the exact position of certain time-dependant elements, such as moving platforms. The space is constrained by screen boundaries and physical obstacles (Σ_{Obs}). A designer may also add other kinds of common platformer elements, such as saws, spikes, *etc.* that kill the player right away, giving us a subspace Σ_{Death}. Moving platforms move in a cyclic fashion along some axis, also limiting player positions, giving us another subspace Σ_{Move} of dynamic obstacles; in figure 2 platform movement is indicated by the grey arrows, with red areas causing instant player death. Given these subspaces, the available movement space is given by

$$\Sigma_{free} = \Sigma - (\Sigma_{Move} + \Sigma_{Death} + \Sigma_{Obs}) .$$

Within the level the player has basic commands to move left-right and jump; we also incorporate double-jumps. Our physics model includes gravitational acceleration downward, but does not include player momentum on the horizontal axis—a player's forward motion depends entirely on the current left/right key-press, and so may be changed arbitrarily and instantaneously, even while in the air (*air control*). We used the 2D or 3D (including time) Euclidean distance for comparing two nodes. From this context we can build a formal model of the game state. We define our space Σ as a combination of subspaces:

$$\Sigma \subseteq \mathbb{R}^2 \times \mathbb{R}^+_{time} \times \mathbb{R}_{fall} \times \{0,1\}_{jump} \times \{0,1,2\}_{moving}$$

This representation encodes the 2D Euclidean space a player may occupy, a non-negative time vector (essential for representing platform movement), a gravity vector to model the jumping or falling velocity, as well as two discrete domains: a 2-valued domain to indicate whether a double-jump has been performed, and a 3-valued domain to represent motion, as either not moving, moving left or moving right.

In our case we have 6 actions, $A = \{$jump, jump-left, jump-right, left, right, wait$\}$. Actions are executed in a discrete fashion for the duration t of a single game frame. The physics is simulated for the period t—in the case of instantaneous actions, such as jump, we assume that after the action is invoked the entity waits (does not do further actions) for t time. The starting state is a particular point in our geometry, at time 0, unmoving, and not double-jumped, and for the goal we use a circular patch of the 2D space of player positions, extruded to a cylinder over the time dimension, and ignoring other components of player state. We represent a player's actual path as an ordered set of actions, separated by a fixed time-step $t' \geq t$.

Using this particular representation it is possible to run multiple paths and present major trends in movement using heat maps, as shown by figure 2. This gives a designer a view of possible locations players may occupy (or not) in the course of solving the game.

Stealth game - We loosely define stealth games as a player task to get from a to b unseen by k enemies. Thus we

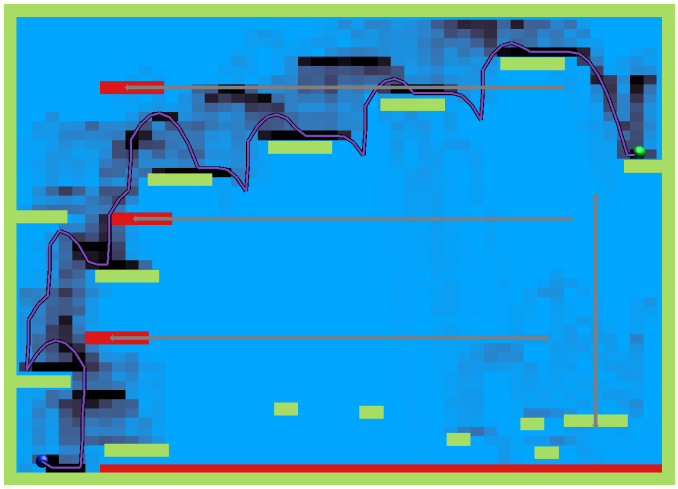

Fig. 2. Heat map showing 1000 paths produced by the framework applied to the platformer, with the darker an area the more a player occupied that cell. A solution is shown in purple, and normal and death areas in green and red, with motion indicated by the gray arrows.

model the basic game space as a finite patch of 2D space, $G \subset \mathbb{R}^2$, representing the underlying 2D terrain, which is then extruded into a third dimension to represent the time evolution of the game, giving us a state space we define as $\Sigma \subseteq G \times \mathbb{R}^+$, assuming a non-negative time dimension. We used 3D Euclidean distance for comparing two nodes, and use non-obstacle space to plan player movements as paths from the initial position to one of the goal position, forming a continuous path within Σ_{free}. Also note that for simplicity of representation we assume player and enemies each occupy a single point on the 2D terrain, although extending our model to give entities actual area of occupancy is straightforward (and is done for visualization below). Hence the state space is defined as follows,

$$\Sigma \subseteq \mathbb{R}^2 \times \mathbb{R}^+_{time}$$

which is simpler than the platformer domain.

Enemies have a fixed field of view (FoV) they use to detect the player, which can be defined as a cone, or more simply as an isosceles triangle, projecting forward from the enemy, and reduced appropriately by geometric occlusions. The goal of the player is then to find a path through the state space, avoiding both obstacles and all enemy FoV. To define the resulting feasible space, we can construct $\Sigma_{fov} \subseteq \Sigma$ as the union of all enemy FoV over time, giving us a final search space defined by $\Sigma_{free} = \Sigma - (\Sigma_{obs} + \Sigma_{fov})$. This is also the space where a (successful) player is free to move.

Figure 3 shows 3 images; (a) is from R. Smith, game designer, describing a hallway with a guard moving back and forth (red line), with the intention that most players should walk following the given blue path, visiting each of the alcoves. Analysis data from our framework, shown in Figure 3 (b), however, tells a different story: players actually only need to use one of the leftmost alcoves to avoid the enemy, and none of the successful paths we found required the player to

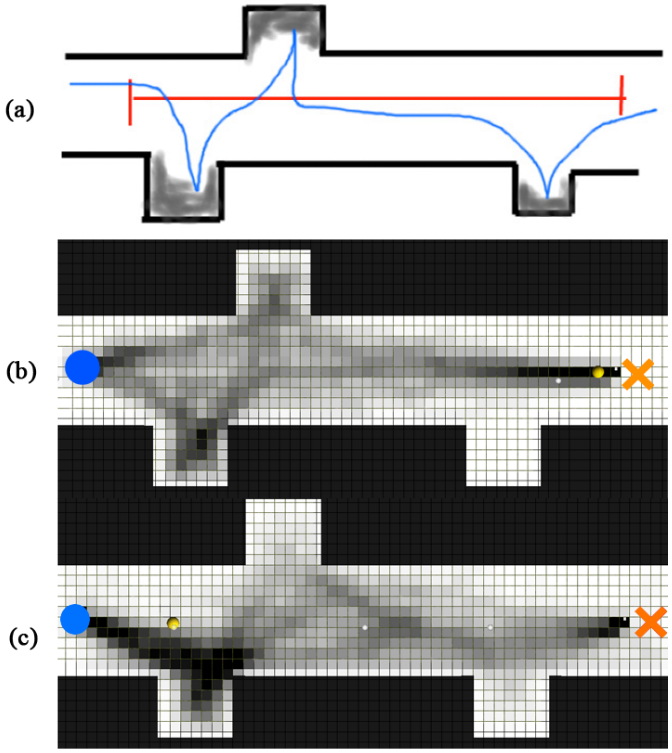

Fig. 3. (a) Level design proposed by Smith [9]. (b) Heat map found from our tool. (c) A guard configuration that makes all alcoves useful.

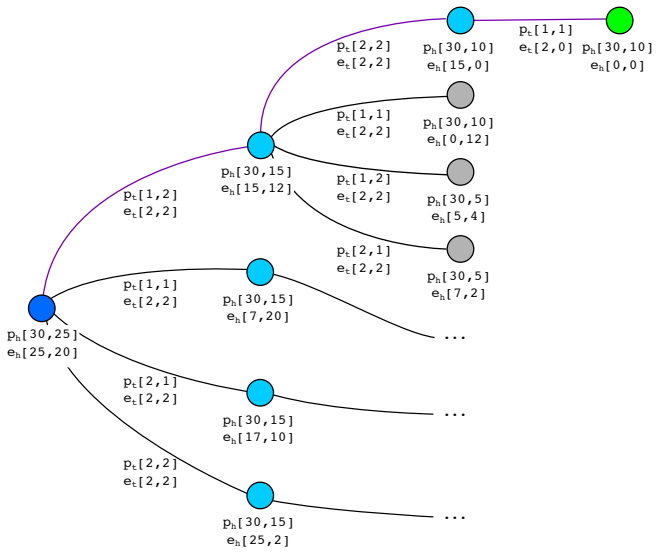

Fig. 4. A path to victory, represented in purple, in an attrition game. Nodes represent game state, and edges are actions. The blue and green nodes represent starting and goal states. Cyan nodes are explored nodes and grey unexplored by the search process. A state is shown as two arrays giving the entities' healths, where $\mathtt{P_h}$ refers to the players' healths and $\mathtt{e_h}$ to the enemies' healths. Following the same approach, an action edge is composed of two arrays declaring the target each entity selected.

use the rightmost alcove to reach the goal. This knowledge can then be used to redesign the guard patrol route so each alcove ends up being used, which can be verified using the design tool as seen in figure 3 (c).

Attrition game - Basic Attrition game are composed of two sides, players and enemies, each seeking to eliminate the other. We define P as the set of players, E the set of enemies, where $|P| = n$, and $|E| = m$. Each entity $p \in P$ and $e \in E$ has attack a and health h where $a, h \in \mathbb{N}^+ \cup \{0\}$. Fighting occurs in rounds, where the players and enemies each select an opposing entity to attack. A player's attack is resolved by deducting $a(p_i)$ from an enemy's health $h(e_j)$; if this leaves $h(e_j) \leq 0$, the entity is killed and cannot take actions in the rest of the current round, or in subsequent rounds. The gameplay is symmetric, although players are assumed to attack first.

Since this is an abstract game, the distance between two state is determined by Manhattan distance on health values,

$$d(\sigma, \sigma') = \sum_{i=0}^{n} |h(p_i) - h(p'_i)| + \sum_{i=0}^{m} |h(e_i) - h(e'_i)|$$

Using RRT, we can then proceed to sample the space with a random distribution of health for both sides, find the closest already explored node using the distance presented before, and extend it toward the sampled state as long as we can find a motion—in this case a combat sequence—that connects states.

Figure 4 shows a victorious path for a simple attrition game.[1] Here there are two players against two enemies: the enemies have 25 and 20 health, and each an attack of 5; players have 30 and 25 health, with 10 and 8 attack respectively; this starting state is shown as the dark blue (leftmost) node. The purple path shows that the players won in 4 turns with both players surviving. This approach allows for pseudo random solutions to an attrition game to be computed, showing possible sub-optimal player paths. From this game developers can sample and observe the variety of possible solutions and evaluate if the attrition game is too difficult or easy.

Discussion & Conclusion

Designing video games can be intensely iterative, with designers refining levels and mechanics based on human play-test data. A framework that leverages state-of-the-art AI search techniques to find (random) game solutions can effectively eliminate most of the labour-intensive aspect of play-testing, improving the design process. To avoid replacing the cost of beta-testing with the cost of search implementation, however, it is important that the approach be sufficiently general, reusing the overall structure and allowing for modular insertion of game-specific structure. Our application to three quite different genres shows that this is both possible and effective.

Our games are implemented within Unity 3D and the tool is designed to be used not only by us but also by game designers. An interesting limitation we encountered in this approach, however, is that the physics simulator is not accessible within

[1]Please note that the syntax used in the figure 3 is different for easier readability.

the editor mode (Unity has two mode, editor and play). This limits the utility to a designer, a problem we addressed by providing our own physics/collision system. Future or more open versions of game engines would simplify this design concern.

Our current focus is on validating the traces we find with respect to real player movements. A more complete model of how closely our search results reflect real players would also allow us to look confidently at unsuccessful, failed player paths, which is also important information to designers.

ACKNOWLEDGEMENTS

A special thank you to Shuo Xu, Pedro Andrade Torres and Alexander Borodovski for helping with the game implementations. This research was supported by the Fonds de recherche du Québec - Nature et technologies, and the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] J. Tremblay, A. Borodovski, and C. Verbrugge, "I can jump! exploring search algorithms for simulating platformer players," in *EXAG 2014: Proceedings of the First AIIDE Workshop on Experimental AI In Games*, 2014.

[2] J. Tremblay, P. A. Torres, N. Rikovitch, and C. Verbrugge, "An exploration tool for predicting stealthy behaviour," in *IDP 2013: Proceedings of the AIIDE Workshop on Artificial Intelligence in the Game Design Process*, 2013.

[3] M. Nelson, "Game metrics without players: Strategies for understanding game artifacts," in *AIIDE 2011: Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2011. [Online]. Available: http://aaai.org/ocs/index.php/AIIDE/AIIDE11WS/paper/view/4114

[4] A. Smith, "Open problem: Reusable gameplay trace samplers," in *IDP 2013: Proceedings of the AIIDE Workshop on Artificial Intelligence in the Game Design Process*, 2013.

[5] A. Jaffe, A. Miller, E. Andersen, Y.-E. Liu, A. Karlin, and Z. Popovic, "Evaluating competitive game balance with restricted play," in *AIIDE 2012: Proceedings of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2012.

[6] M. Shaker, N. Shaker, and J. Togelius, "Evolving playable content for cut the rope through a simulation-based approach," in *AIIDE 2013: Proceedings of the Ninth AAAI Artificial Intelligence for Interactive Digital Entertainment Conference*, 2013, pp. 72–78.

[7] D. Pizzi, J.-L. Lugrin, A. Whittaker, and M. Cavazza, "Automatic generation of game level solutions as storyboards," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 3, pp. 149–161, Sept 2010.

[8] S. M. Lavalle, J. J. Kuffner, and Jr., "Rapidly-exploring random trees: Progress and prospects," in *Algorithmic and Computational Robotics: New Directions*, 2000, pp. 293–308.

[9] R. Smith, "Level-building for stealth gameplay - Game Developer Conference," http://www.roningamedeveloper.com/Materials/RandySmith_GDC_2006.ppt, 2006.