An Algorithmic Approach to Analyzing Combat and Stealth Games

Jonathan Tremblay
McGill University
Montréal, Québec, Canada
Email: jtremblay@cs.mcgill.ca

Pedro Andrade Torres
Salvador, Bahia, Brasil
Email: pedro.torres@mail.mcgill.ca

Clark Verbrugge
McGill University
Montréal, Québec, Canada
Email: clump@cs.mcgill.ca

***Abstract*—Combat and stealth games give players the option of engaging or avoiding enemy agents at different points. Level-design in this context is complex, however, requiring a designer to understand how different design choices impact difficulty under multiple play-styles. In this work we describe a unified algorithmic approach that can perform abstract analysis of both combat and stealth behaviours. Our proposed solution builds on an existing stealth-level analysis tool, incorporating combat activities into the abstraction in order to allow exploration of level feasibility and difficulty. We demonstrate our approach on a non-trivial example level, showing how such a tool can be used to evaluate and control the player experience.**

I. Introduction

Many modern combat games, defined as First Person Shooter (FPS) in the popular literature, combine movement and combat mechanisms. Designing meaningful gameplay experiences for such games involves obstacle placement (level skeleton), enemy design (positioning, combat values, and behaviour), interactive structure, *etc.* Understanding the different possible solutions for such a level is thus not a trivial task, requiring models of movement, combat, and combat-avoidance, and is therefore typically addressed through an expensive prototyping and beta-testing cycle.

We present here an algorithmic approach and its implementation for abstracting and analyzing combat games. This work builds on previous research in modelling stealth games [1], enabling us to analyze level design under arbitrary combinations of stealth and/or combat design objectives. The approach is primarily a problem exploration which aims at practical, design-time exploration of single-player game levels, and naturally accommodates a variety of interesting and common game features, such as combat with multiple enemies and the presence of limited healing resources.

Since no benchmark exists for design-tools, our framework is demonstrated through a non-trivial, *Unity3D*-based implementation. This allows us to analyze game levels in a realistic, industrial-scale game development context, as well as show how such a tool can be used in the actual design process. Using the analysis data generated by our framework, we are able to visualize a variety of metrics such as player movements, deaths, and combats, and show how different arrangement of resources guarantees feasibility, and can shift the balance between places where combat-avoidance is required or not, as well as encourage level exploration. Specific contributions of our work include:

- We extend the analytical domain and search technique used in previous work on stealth analysis to also represent player combat. This non-trivial addition includes a modular representation of combat resolution and results in a unified context for stealth and combat analysis.
- Feasibility of the design is demonstrated by integrating the full framework into Unity3D.
- We demonstrate the value of our approach on a representative game level, showing how the tool can expose useful stats on the level and help guide a level designer to an appropriate design.

II. Background & Related Work

We are interested in simulating artificial players for two game genres: combat and stealth games. We loosely define combat games by a player getting from *a* to *b alive*. The player has to survive every *combat* against other Non-Player Charaters (NPC), with combat initiated when the player walks into an NPCs' Field of View (FOV). This definition includes FPS games such as *Half-Life* or Role Playing Games (RPG) such as *Baldur's Gates*. Stealth games can then be seen as a subset of combat games, where combat is disallowed, or at least not required: the player has to avoid each NPC's FOV while getting from *a* to *b* [2]. Pure examples of this genre exist, such as in the *Thief* series, but many and more recent games in the stealth genre, such as *Dishonored* allow some amount of combat, and a combined consideration of both combat and stealth behaviours is essential to their design and understanding.

Although distinct game genres, both combat and stealth behaviours are fundamentally based on the task of path-finding from *a* to *b*, and we can understand player behaviour as based on searching a complex space, including enemies for feasible paths. To simulate player behaviour, different algorithms exist that would search different space representations to find a path between two positions. For example, Dijkstra, A*, Rapidly Randomly exploring Tree search (RRT), *etc.* [3]. Our work focuses on RRT, which we define in section III; for a better understanding of path-finding in general the reader is referred to Klingensmith [4].

Our design relies on being able to determine the results of combat encounters. Simulating combat is a non-trivial task, however. Normal combat scenarios in FPS games involve many moving parts, such as physics, different attack and defence attributes, geometries, *etc.*, and a full consideration makes

accurate simulation complex. In RPGs, there exists simpler combat system (in terms of simulation) where the player and enemies take turns choosing actions and targets. Unfortunately, optimally solving even relatively simple discretized-time combat games, such as Basic Attrition Games, has been previously shown to have exponential time complexity [5], and even the decision problem is PSPACE-hard. We thus do not attempt to determine optimal combat resolution, and instead rely on a simple "damage per second" (dps) computation based on health and enemy attack values. Dps is widely used as a meaningful threat measurement of an entity in games such as *World of Warcraft*.

A. Related Work

The work we describe in this paper can be seen in the context of several research efforts which use AI techniques during the game creation process in order to understand games and its design. As Nelson describes, "We need not treat the game as a black box [...] we can analyse the game itself to determine how it operates" [6].

Different approaches have been taken as to apply AI to game design. Jaffe *et al.,* for instance, explore the concept of *fairness*/game-balance in games based on win-rate [7]. 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. In the game *Cut the Rope*, they discretized time, finding the next game state by using a physics simulator to update the world [8]. From this update, the available actions are then explore using depth-first search. This allowed them to present a path in a tree of game actions that solves the level. This work is comparable to our presented work in discretizing a complex state space, although using different search algorithms and aimed at a very different game context.

More specifically aimed at simulating stealth games, Pizzi *et al.* abstracted the level structure and gameplay of the popular video game *Hitman: Blood Money* into discretized events that could be assembled into storyboards [9]. These storyboards represent major actions such as walking to a point, taking down an enemy, procuring an item, *etc.* This abstraction allowed the designer to build a restricted level (placing enemies and items), where a planner would find a path to the winning position. Following the sketching tool concept, Liapis *et al.* presented a tool where designers could draft levels using a high-level terrain editor which included different metrics such as *symmetry, area control,* and *exploration* [10].

Once traces are obtained, whether artificially generated or not, visualization is essential to the design process. A basic approach to this problem is to use different heat maps or influence maps, such as is commonly done to model player death positions [11]. Understanding the design space is also achievable using feature-based state projections [12] or by measuring different metrics on the paths such as difficulty [13].

III. Tool Design

Our work is based on abstracting the game state space into a high-dimensional geometric space, and then using path-finding techniques to explore reachability. For this we extend an existing framework that used a similar technique, but which used a smaller state space to target only stealth-game levels [1]. Below we first present a formal abstraction of the space, the search process, and combat simulation, followed by a discussion of different approaches to visualizing the results.

A. State space

Our representation is based on a 2-dimensional game terrain, wherein a set of k enemies E and a single player interact over time. Enemy movements in the absence of player interaction are assumed to be deterministic, and so we can compute an enemy's state (x, y position and orientation) for any given time from the pre-determined game level specification. Both enemies and the player have scalar health values as well as (fixed, pre-determined) attack values to represent the damage done by a single attack. For enemies the health values will be either at maximum or 0, while for the player the health value may be any number—this is meant to allow us to model game-state before and after combat, with the assumption that combat always terminates with either the enemy dead and player at partial health, or the player dead (simulation over), and so there is no need to explicitly model the behaviour of enemies with partial health.

To represent this we build on a model of the game space as a patch of $\mathbb{R}^2$. This space is extruded over time to represent the evolution of the game during gameplay, giving us a 3D domain $\Sigma \subseteq \mathbb{R}^2 \times \mathbb{R}^+$ (assuming only non-negative time values). The reachable states in this space exclude obstacles, and so we will generally be more interested in $\Sigma_{free} = \Sigma - \Sigma_{obs}$, where Σ_{obs} is the space occupied by the level obstacles. Incorporation of the player and enemy states in this space adds other dimensions, giving us a full state space of $\Sigma \times \mathbb{R} \times \mathcal{P}(E)$, where we include a real value for player health, and a subset of living enemies. Note that more complex representations, such as a 3D game terrain, or including variability in damage dealt (depending on player health) are straightforward to accommodate through additional dimensions.

Within the state space we are interested in specific gameplays, or *paths* that the player may take through the state space. Paths are in general continuous functions on the time dimension, mapping monotonically increasing time values to elements of the rest of the state space: $\rho : \mathbb{R}^+ \rightarrow \mathbb{R}^2 \times \mathbb{R} \times \mathcal{P}(E)$. Feasible paths must of course respect movement constraints for the player, enemies, and any other relevant game mechanics (no movement through obstacles, dead characters stay dead, *etc.*), and will be constructed in a discrete fashion, as a set of connected points in the state space. Given a point in a path $\sigma \in \rho$, we will access specific data through the use of simple projection functions: $\text{pos}(\sigma) =< x, y, t >$ retrieves the position of the player at time t, $\text{alive}(\sigma) \subseteq E$ returns the set of live enemies, and $h(\sigma) = h$ returns the player's health. Although not part of the state space itself, we use a for the player attack value, $h(e)$ to access an enemy's health, and $a(e)$ to access an enemy's attack value.

Enemies also follow paths through time, interacting with the player only when they see her. Enemy motions are deterministic, and so in order to understand whether there will be any player interaction we mainly need to be able to retrieve each enemy's FOV at any given point in time. In our simplified

environment, enemies' FOV at each instant are modelled as 2D cones, with the enemy located at at the apex. Enemy visibility over a span of time can then be represented by a layering of instantaneous FOVs forming a collection of triangular prisms, slanted in relation to the enemy's movement speed, joined by slightly more complex polyhedra during rotations, and constrained by obstacle occlusion. We use $g(e)$ to access the geometric shape that describes an enemy's complete FOV for the duration of the entire simulation.

B. Basic path-finding

Construction of player paths in our multi-dimensional space can be performed through different path-finding algorithms. We prefer the use of *a Rapidly Exploring Random Tree* (RRT) to other algorithms for its practical efficiency and its inherent variability, as we are less interested in finding the optimal paths and more in approximating the variety of possible paths human players may take.

In order to find a path RRT builds a tree starting from a given initial state, σ_{init}, trying to reach a given goal state, σ_{goal}. We treat the latter as a set, since in general achieving close proximity to the goal is sufficient. In the absence of enemies and combat this process is straightforward, and is illustrated in figure 1 (this figure also contains additional complexities, which we will describe in the following subsections). We assume an existing tree-structure, Υ, of states σ is maintained, initially consisting of just σ_{init}. Expansion of the tree then proceeds by randomly chosing a new point in the feasible state space, $\sigma_{rand} \in \Sigma_{free}$. This is shown as the green node in figure 1. We next find the nearest point $\sigma_{near} \in \Upsilon$ to σ_{rand} using an appropriate distance metric, such as Euclidean distance. This is shown as the blue node in figure 1. We then verify if it is possible to reach σ_{rand} from σ_{near} within Σ_{free}. If the segment $(\sigma_{near}, \sigma_{rand})$ is collision-free then the segment and node are added to Υ, growing the tree by one branch; otherwise we discard σ_{rand}. This process is then repeated until we encounter a σ_{rand} contained in σ_{goal}, in which case the search has succeeded, or we reach a limit in time or tree-size, in which case the search fails. In the former case we form a path $\rho \in \Upsilon$ by tracing a path through the tree. For a deeper look at RRT the reader is referred to Morgan *et al.* [14].

C. Incorporating combat

Basic path-finding using just the 3D $(x, y, \textit{time})$ subset of our domain is already useful for modeling stealthy behaviours, with previous work finding stealthy paths that avoid combat altogether by simply treating the deterministic enemy FOVs as obstacles in a 3D state space [1]. The complete algorithm is shown as algorithm 1, with bold line numbers used to identify our new extensions. Consideration of the combat dimensions to the model, however, not only expands the state space, it greatly complicates the search, as we now need to consider the existence of enemies, combat state, and player health in selecting and attaching new nodes to the search-tree—the feasibility of a path from an existing node to a randomly selected node requires non-trivial modelling of combat behaviours, and is not simply determined by the movement model. Rather than just naively apply RRT to this extended space, we thus use a variant on the basic RRT, where the node attached to the tree at each iteration is determined computationally from the initial, random choice. We show the resulting pseudo-code in algorithm 1, and describe it below. Note that to reduce complexity in this exposition, we assume that only one enemy can fight the player at a given time and that the player cannot lose a fight; these constraints are relaxed in the next subsection.

Our approach retains the core strategy of randomly selecting reachable points in the state space in order to grow a search-tree. To make this selection we ignore player health and enemy liveness, so this process starts off the same as for the basic RRT search. In determining whether σ_{rand} can be connected, however, we may discover that the connecting segment intersects an enemy FOV. If this enemy is still alive in σ_{near} then combat will occur if the player follows that path segment. The COLLISIONENEMY function called on line 8 of algorithm 1 shows this determination. In this function (lines 27–39), we use the helper method LINEPRISMCOL to compute the point closest to σ_{near} in the intersection of $(\sigma_{near}, \sigma_{rand})$ and each living enemy's FOV structure. The function returns the full state at which combat will initiate, σ_{combat}, shown as the red node in figure 1, as well as the enemy involved.

Having found a point at which combat occurs, the algorithm simulates the result of the combat based on a damage-per-second (dps) calculation. This is performed by the SOLVECOMBAT function shown on lines 41–48. Here we simply divide the enemy health by player attack to give a duration for the combat, and use that duration to also determine how much health the player loses. Combat is assumed to occur in a single location, giving us a result state, σ'_{rand}, as a state projected upward in time from σ_{combat} with updated player health and enemy sets, shown as the purple node in figure 1. The search-tree is then updated by adding σ_{combat} and σ'_{rand} to Υ through segments $(\sigma_{near}, \sigma_{combat})$ and $(\sigma_{combat}, \sigma'_{rand})$, representing the player reaching the point of combat, and engaged in combat respectively. Note that this adds two nodes to the tree, although combat-initiation nodes such as σ_{combat} cannot be nearest neighbours for future σ_{rand} choices. In cases where no enemies can be encountered we follow the basic RRT algorithm, adding valid and reachable states to the tree.

Finally, when a goal state is reached, the PATH$(\Upsilon, \sigma_{rand})$ method call on line 18 retraces a shortest path, ρ from the last node added to the initial state, and we add ρ to our collection of paths P. The entire process is iterated and controlled through two user-defined parameters to specify how many paths the user wants to see (M), and how large she wants the tree to grow (N).

D. Player death and overlapping combats

In using the above algorithm in the context of design exploration, it is extremely useful to know not just where and how a player may succeed, but also where players may be unsuccessful and have died. To enable this information to be retained, we drop the requirement that the player always wins combat, and allow states with $h(\sigma) \leq 0$ to be added to the tree. This adds a minor complication, in that when finding the nearest state to σ_{rand} we also have to make sure that the player is still alive.

Integration of combat with multiple enemies is equally easy. Multiple enemy combat occurs when another enemy is

Algorithm 1 Pathing in our high-dimensional space. Bold line numbers indicate code associated with our extensions to previous work [1].

```
 1: procedure CompPaths(σ_init,σ_goal,Σ_free,N,M)
 2:     for i = 0 to M do
 3:         Initialize(ϒ, σ_init)
 4:         for j = 0 to N do
 5:             σ_rand ← Sample(Σ_free)
 6:             σ_near ← Nearest(σ_rand, ϒ)
 7:             if CollisionFree(σ_near, σ_rand, Σ_free) then
 8:                 σ_combat, e ← CollisionEnemy(σ_near, σ_rand))
 9:                 if σ_combat ≠null then
10:                     σ'_rand ← SolveCombat(σ_combat, e)
11:                     ϒ ← (σ_near, σ_combat)
12:                     ϒ ← (σ_combat, σ'_rand)
13:                     σ_rand ← σ'_rand
14:                 else
15:                     ϒ ← (σ_near, σ_rand)
16:                 end if
17:                 if σ_rand ∈ σ_goal then
18:                     ρ ← Path(ϒ, σ_rand)
19:                     P ← ρ
20:                     break
21:                 end if
22:             end if
23:         end for
24:     end for
25: end procedure
26:
27: procedure CollisionEnemy(σ_near, σ_rand)
28:     σ_combat ←null, e ←null, d ← ∞
29:     for each e_i ∈ alive(σ_near) do
30:         v ← LinePrismCol(σ_near, σ_rand, g(e_i))
31:         if v ≠ null ∧ |(pos(σ_near), v)| < d then
32:             σ_combat ← σ_near
33:             pos(σ_combat) ← v
34:             d ← |(pos(σ_near), v)|
35:             e ← e_i
36:         end if
37:     end for
38:     return σ_combat, e
39: end procedure
40:
41: procedure SolveCombat(σ_combat, e)
42:     σ'_rand ← σ_combat
43:     Δt ← h(e)/a
44:     t(σ'_rand) ← t(σ_combat) + Δt
45:     alive(σ'_rand) ← alive(σ'_rand) − e
46:     h(σ'_rand) ← h(σ_combat) − a(e) · Δt
47:     return σ'_rand
48: end procedure
```

able to observe the combat between the player and an enemy, and so joins in the attack on the player. In our model this is true if the segment $(\sigma_{combat}, \sigma'_{rand})$, representing combat with enemy e intersects the FOV of a different, living enemy, e'. If so, we assume the player continues to fight e to completion, queueing the combat with e' until the fight with e completes. This has the effect of revising σ'_{rand}, extending combat to last the full duration of $(h(e) + h(e'))/a$. As well as projecting σ'_{rand} further in time, we also update σ'_{rand} to reflect the death of the additional enemy, and to include the additional health loss to the player of being attacked by both enemies during the overlap of combat with e and e', and which was suffered during

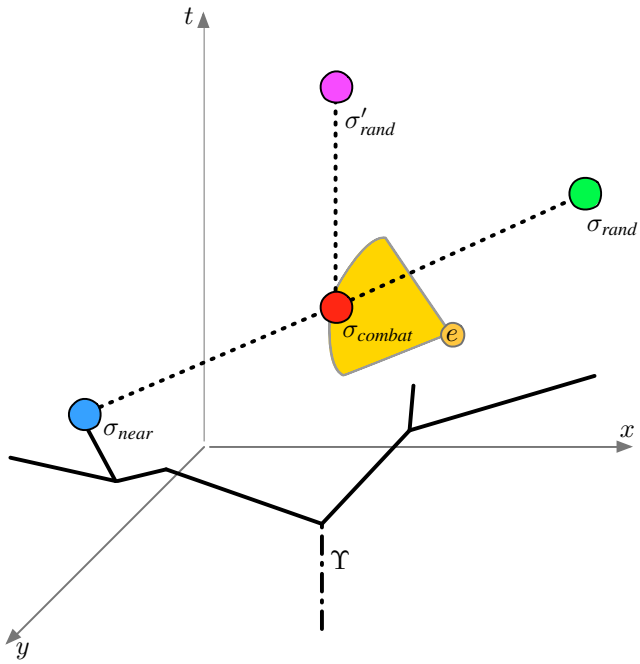


Fig. 1: Combat algorithm representation

the queued combat with e'. This process has a straightforward extension to accommodate an arbitrary number of enemies within the same combat sequence. It is also an approach that is quite modular in the detail of combat representation, and although we have used a very simplistic calculation for combat outcome, integration of more elaborate models that optimize combat choices [15] is part of our future work.

E. Visualization and analysis

Appropriate representation and analysis of the paths found is important for practical use of this approach in level analysis and design. Examining any single, given path is straightforward: every state has a time component, so we can visualize the path by replaying the player and enemy motions in real-time. Figure 2 shows a snapshot of this approach from our implementation, using a blue sphere for the player, yellow spheres for living enemies, line segments to indicate the player's historical path (blue) and current combat engagement (red), and terrain colourization to show enemy FOVs (based on a discretized space model). In this way we can see the exact progress and behaviour of a possible player path.

Unfortunately, this approach does not scale well to representing multiple paths. Not only do many paths clutter the view, paths in general have different sets of live enemies, making it difficult to determine which set of enemies is relevant to each path. For understanding multiple paths, we thus make use of simple heat maps, flattening all paths in the time dimension, and shading terrain cells according to the density of path points with the same x, y coordinate. Applying this technique to show player movements can be quite illustrative, revealing locations or enemy positions which successful player paths avoid or commonly target.

Of course, and by intention, the paths generated by our search process constitute only successful gameplays—the

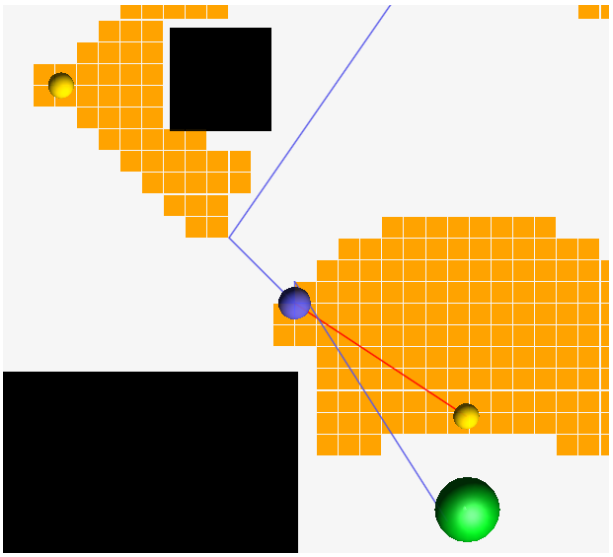


Fig. 2: A player path (blue line) displayed in Unity 3D, with a player (blue sphere) fighting (red segment) with an enemy (yellow sphere).

player reaches the goal state. Designers, however, are also interested in where combat occurs, and in particular where players are likely to fail (die). To display the latter information we need to extract data from the nodes attached to all search-trees, and not just the successful paths. When a state is added to the tree, we thus check if $h(\sigma'_{rand}) \leq 0$, and if so we save $pos(\sigma'_{rand})$ in a data structure. We then display this information as a separate heat map, showing the relative density of player deaths at each location. Since we have time data in these points as well, this heat map can also be examined as a real-time replay, displaying deaths as animations over an interval $t \pm \alpha$ for some α. A similar process can be applied to create visualizations of combat or enemy deaths.

Finally, we use our analysis framework to also gain some basic knowledge about the difficulty of the level. The ratio of successful paths found versus searches performed ($|P|/M$) gives a measure of how easy it was for the RRT to find a solution, and thus a heuristic indication of how difficult players may find the overall experience. We can also easily retrieve other, more detailed path information, such as the time taken to complete the level, how many enemies where killed, how many player-death states were added in the search, how much health was lost, and so forth. In the next section we demonstrate use of these different visualizations and data assessments.

IV. Experiments & Results

In this section we will explore different applications of the presented framework. We first present a simple case scenario that demonstrates basic usage and results, and then apply the technique to a more complicated context with multiple enemies, and where we introduce health packs. For every test, 1500 searches were performed, with each search capped at sampling 25000 states; *i.e.,* $N = 1500$ and $M = 25000$ in algorithm 1. Note that our implementation is based on a discretized version of the state space described in the previous section.

A. Simple level

Our first test illustrates how the search process can traverse combat situations and help identify situations in which combat is possible or must be avoided. For this we use three different parametrizations of the same level, heat map results for which are shown in figure 3. In this level the player starts at the blue sphere and has to reach the green sphere, pathing around an obstacle and possible encountering a single, statically positioned enemy on the north side.

In figure 3a the enemy has a health of 10, while the player has infinite health. This makes combat trivial, and we see that paths both pass through the enemy FOV, and thus engage in combat, and bypass the enemy: both combat and stealth solutions are viable. An interesting variation is shown in figure 3b, where we give the enemy 200 health. This results in a longer combat sequence (average path length increases from ~1000 time steps to ~1200), but since our search process does not favour or optimize for faster paths no significant difference in the resulting movement heat maps is evident.

If player health is limited we expect the stealthy option to be strongly preferred. This is shown in a limiting sense in figure 3c, where we give the player just one health unit, and the enemy 10. This makes successful combat impossible for the player, and indeed the heat map of movement shows only paths that avoid the enemy. Interestingly, these are the fastest paths at just ~500 time steps on average.

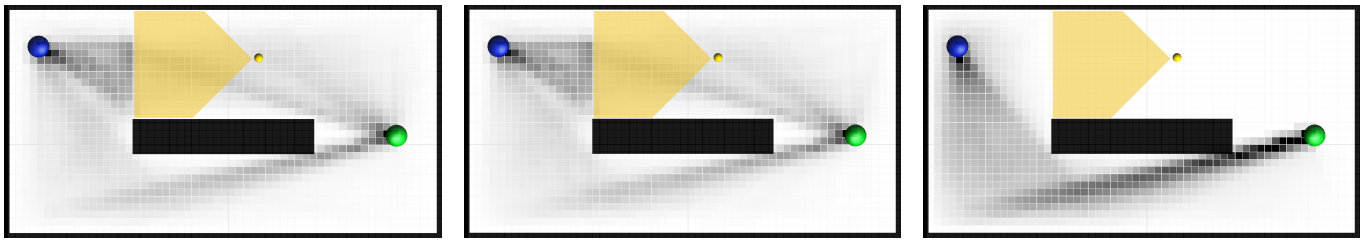
B. Non-Trivial Level

In this subsection we explore a non-trivial level with four enemies, including a *boss*. Figure 4 maps out the setup; the player must traverse a fairly large space containing 3 patrolling enemies with 33 health and 3 attack each; enemies e_1 and e_3 have the most complicated paths, covering a significant part of the level. Enemy e_4 blocks the exit of the level and acts as a boss fight, with 99 health and 10 attack. The player has 100 health and 10 attack. Note that paths do sometimes overlap, and so it is possible that the player may need to fight two enemies simultaneously. This level is designed in such a way that the player can only reach the exit if she has full health, and fighting any of e_1, e_2, e_3 will not leave enough health to complete the boss fight (and vice versa, fighting the boss will not leave enough health to fight any minion).

The search process found 386 successful paths out of our 1500 searches, and figure 5a shows the resulting movement heat map. Numeric data in table I verifies that these paths all involved a single enemy combat (with e_4). The location of player deaths and combats also supports this interpretation. Figure 5b shows where players died, which was as expected primarily against e_4, while figure 5c shows that in general combat happened along the most straightforward route used to reach the goal.

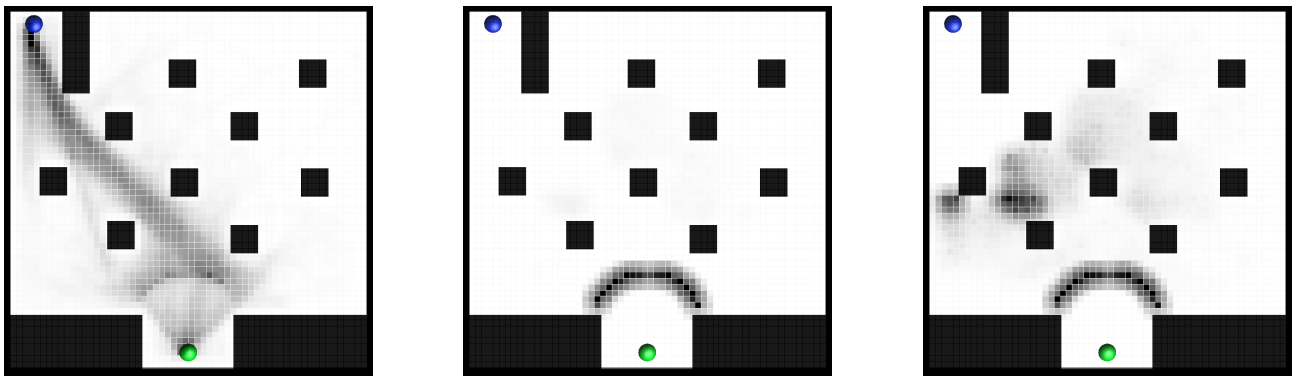
C. Adding resources

From a game designer perspective, the non-trivial level might seem overly biased to stealth, as the player has to avoid all moving enemies in order to reach the goal. In many FPS games the player is given more flexibility in choice through the inclusion of supportive resources, such as health packs or



(a) Enemy with health 10 (b) Enemy with health 200 (c) Player with health 1

Fig. 3: Movement heat maps for different enemy and player health parametrizations. The player starts at the blue sphere and has to reach the green sphere. A single enemy is represented by the yellow dot and yellow FOV.



(a) Movement of 386 paths found (b) Death locations of 9262 dead states (c) Combat locations

Fig. 5: Non-trivial level

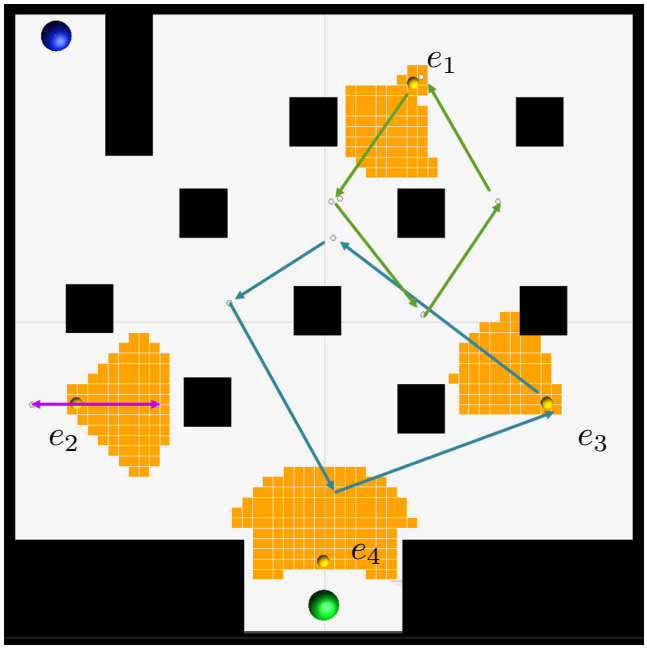


Fig. 4: Enemy paths for the non-trivial level.

ammo. Our framework easily extends to include such basic resources. For example, in order to incorporate health packs, we simply add another dimension to our state σ to specify still available health packs' position. We also modify movement to represent player decisions to seek out health packs—when adding a state σ to the tree, we check if $h(\sigma) < 100$ and a health pack is reachable without requiring combat. If so, we create a new node σ_{health} with the player's health updated by the health pack's value and the used health pack removed from the set of all health packs. Then the new node σ_{health} and segment $(\sigma, \sigma_{health})$ are added to Υ. A similar process could be applied to other resources such as ammo.

Inclusion of health packs modifies the search results, although the resulting behaviour also depends on where the packs are located. In figures 6 through 8 we show results for different placements and numbers of health packs, indicated visually by green squares. Table I gives the corresponding numeric data.

Far health pack - In order to improve the level design, we first included a health pack near e_1. This is relatively far from the player's typical path, but allows a player to fight even all 3 minions (e_1, e_2, e_3) and then pick up the health pack before fighting the boss. Although this scenario is unlikely, introducing the health pack facilitates the completion of the level as only 1276 dead states were found compared to 9262 in the base non-trivial level. In figure 6a we can see 474 successful paths found out of 1500, where about 60% of

TABLE I: Numeric data for different level designs

		No health pack	Far health pack	Near health pack	4 health packs
Total paths		386	474	969	982
Enemies Killed	0	-	-	-	-
	1	100%	42 %	7 %	5 %
	2	-	5 %	56 %	23 %
	3	-	25 %	30 %	32 %
	4	-	28 %	7 %	40 %
Player deaths		9262	1276	1616	713

them used the health pack. Comparing the death locations of figure 5b to figure 6b, we can observe that the player still tended to die near e_4, although there is a significant reduction in quantity (by a factor of ~7). Including a health pack also shifts the combat behaviour. In figure 6c we now see more combat around e_1's position, and from the data in table I we can also see that most of the paths now involve fighting 3–4 enemies, although almost half of the solutions still fight only one enemy, and so are not making good use of the health pack.

Close health pack - A designer is most likely going to put a health pack where players tend to fail. In figure 7 we thus place it near e_4. This positioning results in 969 paths out of 1500 being successful, a significant improvement over the no and far health pack approaches. Interestingly, the number of player death states is greater than with the far health pack; this is explained by the limit on searchable states, N, which in the case of the far health pack was reached sooner since it encouraged greater area exploration. Here the distribution of enemies killed changes again, with most of the paths fighting 2 enemies, e_2 and e_4 as indicated by figure 7c. Again, however, most player deaths were found in front of the boss enemy.

Four health packs - Having only one health pack was perhaps restrictive, as the player had to plan when to use the health pack, which was best done just before fighting e_4. To simplify this, we added four health packs in the middle of the level. This reduces the need for a singular strategy, and also makes the level easier to win. As table I shows, the number of player death states is again reduced, while maintaining the large number of successful paths found with the near health pack. Visual results are shown in figure 8. Fights are now more likely to happen around the locations of the health packs, and in figure 8b, we can observe that e_4 is no longer the primary source of player failure, switching instead to a location where combined combat with e_1 and e_3 is likely. Adding the four health packs also lets the player move more freely; table I shows that over 70% of paths kill three or four enemies.

V. Discussion & Conclusion

In this work we described an algorithmic approach for finding feasible paths that takes into account combat potential and results, in addition to the possibility of combat avoidance. This gives us a broad model of possible player behaviours, as an essential part of understanding and exploring level properties in combat/stealth games. Our framework is useful for gaining deeper understanding of level design, and easily extends to handle simple resources management, *e.g.* health packs.

For future work we are interested in improving the veracity of the combat simulation, as well as exploring and validating different approaches to ensuring our paths reflect real player behaviours. This includes modelling adventure or massively multi-player game combat systems. Different RRT controllers (such as avoiding combat if health is lower than a given threshold) may better correlate with actual player traces, while still giving us a fast, analytical, design-time approach to understanding game levels.

Acknowledgements

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

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

[3] I. Millington and J. Funge, *Artificial Intelligence for Games*, 2nd ed. Morgan Kaufmann, 2009.

[4] M. Klingensmith, "Overview of motion planning," 2013, http://www.gamasutra.com/blogs/MattKlingensmith/20130907/199787/.

[5] T. Furtak and M. Buro, "On the complexity of two-player attrition games played on graphs," in *AIIDE-2010: Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2010.

[6] M. J. Nelson, "Game metrics without players: Strategies for understanding game artifacts," in *IDP 2011: Proceedings of the 2011 AIIDE Workshop on Artificial Intelligence in the Game Design Process*, 2011, pp. 14–18.

[7] 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, pp. 26–31.

[8] 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.

[9] 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.

[10] A. Liapis, G. N. Yannakakis, and J. Togelius, "Towards a generic method of evaluating game levels," in *AIIDE-2013: Proceedings of the Ninth AAAI Artificial Intelligence for Interactive Digital Entertainment Conference*, 2013, pp. 30–36.

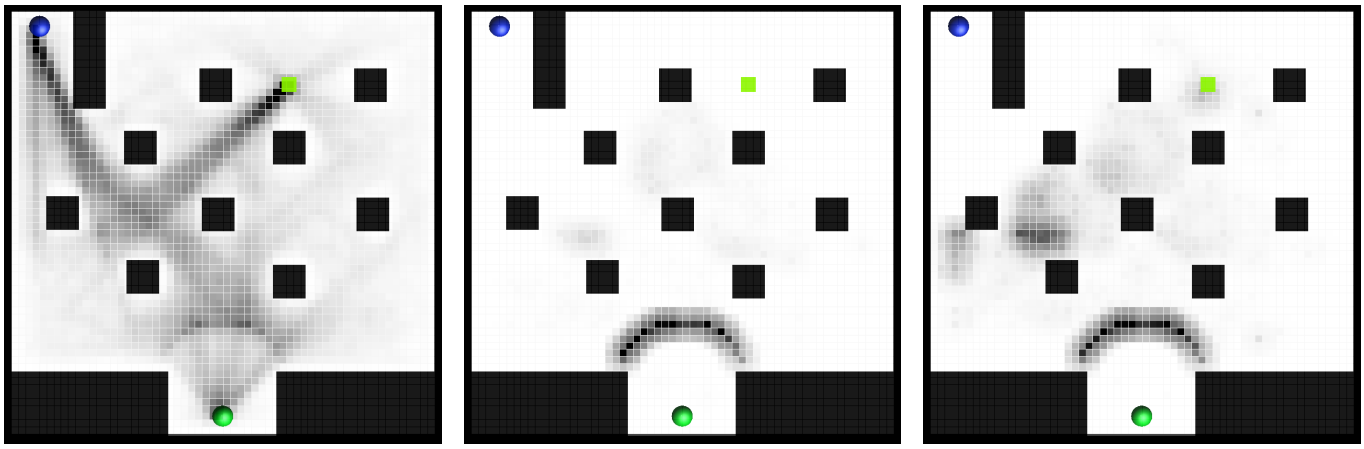
[11] J. Hagelbäck and S. J. Johansson, "A multiagent potential field-based bot for real-time strategy games," *International Journal Computer Games Technology*, pp. 1–10, 2009.

[12] Y.-E. Liu, E. Andersen, R. Snider, S. Cooper, and Z. Popović, "Feature-based projections for effective playtrace analysis," in *FDG 2011: Proceedings of the 6th International Conference on Foundations of Digital Games*, 2011, pp. 69–76.

[13] J. Tremblay, P. A. Torres, and C. Verbrugge, "Measuring risk in stealth games," in *FDG 2014: Proceedings of the 9th International Conference on Foundations of Digital Games*, 2014.

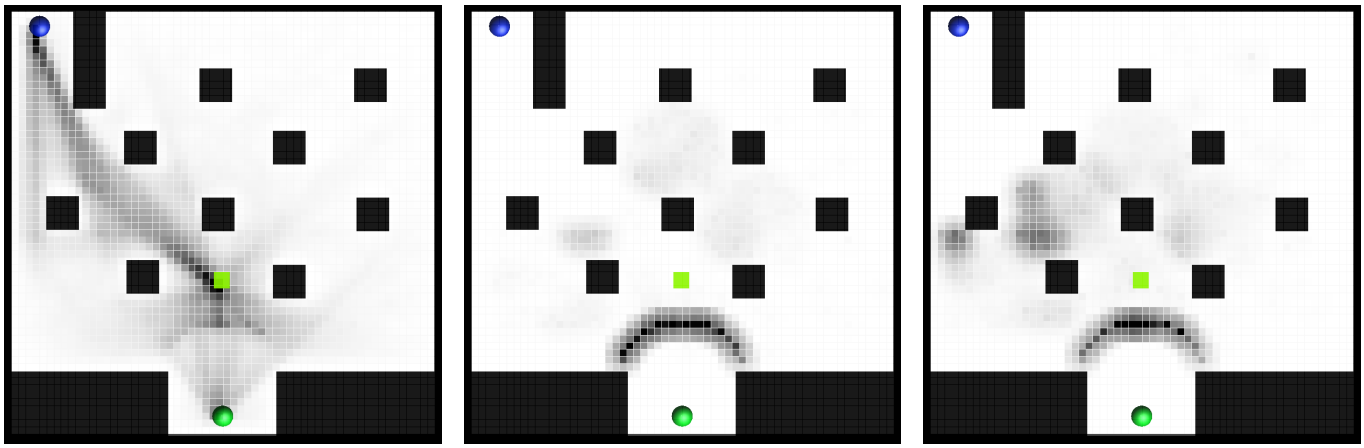
[14] S. Morgan and M. Branicky, "Sampling-based planning for discrete spaces," in *Proceedings of the International Conference on Intelligent Robots and Systems (IROS 2004)*, vol. 2, 2004, pp. 1938–1945.

[15] J. Tremblay, D. Christopher, and C. Verbrugge, "Target selection for AI companions in FPS games," in *FDG 2014: Proceedings of the 9th International Conference on Foundations of Digital Games*, 2014.



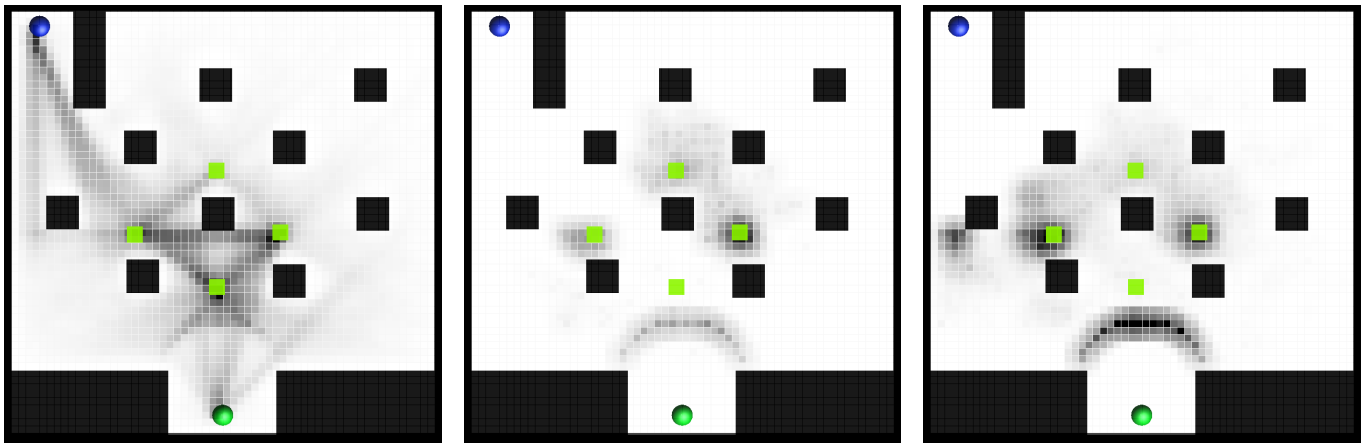
(a) Movement of 474 paths found (b) Death locations of 1276 states (c) Combat locations

Fig. 6: Non-trivial level with health pack (green square) near e_1



(a) Movement of 969 paths found (b) Death locations of 1616 states (c) Combat locations

Fig. 7: Non-trivial level with health pack (green square) near e_4



(a) Movement of 982 paths found (b) Death locations of 713 states (c) Combat locations

Fig. 8: Non-trivial level with 4 health packs (green squares)