An Exploration Tool for Predicting Stealthy Behaviour

Jonathan Tremblay*, **Pedro Andrade Torres***, **Nir Rikovitch**† **and Clark Verbrugge***

*School of Computer Science
McGill University
Montréal, Québec, Canada
jtremblay@cs.mcgill.ca
pedro.torres@mail.mcgill.ca
clump@cs.mcgill.ca

†Department of Mechanical Engineering
McGill University
Montréal, Québec, Canada
nir.rikovitch@mail.mcgill.ca

Abstract

Stealthy movement is an important part of many games in the First Person Shooter (FPS) and Role Playing Games (RPG) genres. Structuring a game level to match stealth goals, however, is difficult, and can depend on subtle and fragile interactions between the game space, enemy motion, and other factors. Here we apply a probabilistic path-finding approach to efficiently analyze a 2D space and find stealthy paths. This approach naturally accommodates variation in the level design, numbers and movements of enemies, fields of view, and player start and goal placement. Our design is integrated directly into the *Unity 3D* game development framework, allowing for interactive and highly dynamic exploration of how different virtual spaces and enemy configurations affect the potential for stealthy movement by players, or other NPCs.

Introduction

Stealthy movement is a critical component of gameplay for many games. With respect to players, the existence of stealth-based approaches provides variety and further strategic choices in approaching combat. NPCs also benefit from understanding stealth, as failure to recognize and follow a player's sneaking behaviour can either interfere with player strategy by accidentally inducing combat (a frequent problem in *Skyrim* (Bethesda Game Studios 2011)), or result in immersion–breaking visual artifacts as enemies fail to acknowledge overtly non-stealthy NPC movements when the player enters stealth mode (such as in *The Last of Us* (Naughty Dog 2013)).

Ensuring an interaction scenario is amenable to stealthy movement, however, is quite difficult. The existence of movement paths appropriate for sneaking depends on the complex interplay between enemy senses, enemy placements and movements, the location and occlusions provided by virtual objects, and player starting and goal states. Tools which allow for interactive exploration of how the potential for stealth is affected as these factors are modified thus have obvious value, enabling better level design and providing trace data or online mechanisms for improving NPC behaviours when they accompany a sneaking player.

In this work we design, implement, and test a tool for visualizing and exploring stealth paths in a 2D game level. Our design is based on known AI techniques, tailored for efficiency, and adapted to the specific task of interactively understanding sneaking behaviours. By integrating our tool into *Unity 3D* (Unity Technologies 2013), a full featured, industry relevant tool for game development, we enable efficient exploration during the game design phase, accelerating the process of prototyping and early-stage verification (Nelson and Mateas 2009).

We verify our efforts on two non-trivial examples, as well as detailed performance analysis. The first example is motivated by industry discussion of basic idioms in sneaking design (Smith 2006), and demonstrates the complexity of even these simple interactions. A second example tests a more involved context, reproducing a level design from *Metal Gear Solid* (MGS) (Konami Digital Entertainment 1998), and is intended to show our tool design scales to realistic contexts.

Background & Related Work

Use of stealth in games can be inherent to the game design, although it is more typically part of a separate, but coexisting form of gameplay with a distinct movement and interaction model (*i.e.,* a separate stealth mode). In either case a stealth game (or level) is simply defined in terms of presenting a challenge for the player to get from one location to another, while avoiding detection by static and mobile enemies.

The challenge of being stealthy can then be mitigated through different game mechanics. In-game tools may be provided to the player to aid in sneaking, such as use of the cardboard box in the *Metal Gear Solid* (MGS) series, arrows in the *Thief* series (Looking Glass Studios 1998), as well as special abilities, such as teleportation in *Dishonored* (Arkane Studios 2012). Gameplay and challenge may also be influenced by environmental factors: snow may leave visible movement traces, metal floors or loose objects can be noisy on contact, and light or shade may alter visibility positively or negatively. When designing a level a designer has to take these variables into consideration, and combine them with the basic properties of enemy position and movement and detection abilities in order to build the right experience. Smith defines a level to be *stealth friendly* if the in-game tools that reduce detection are greater than the environmental challenges that increase it (Smith 2006).

Stealth is also encountered as part of normal combat preparation, where players seek to scout out the environ-

Copyright © 2013, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

ment in order to gain knowledge about enemy movements and placements. This usage constitutes less of a stealth game in itself, but does not change the main techniques involved.

Level Design Tools - Our approach in this work is to develop a tool to assist in stealth level design. Tool development is of course an important and large aspect of game research and development, and at its highest level includes full frameworks, such as *UDK* (Epic Games 2008) and *Unity 3D* (Unity Technologies 2013). Here we are interested specifically in tools that help the designer understand his or her designs. These kinds of tools use AI techniques that focus on how the inner structure of the game operates and structures the play experience.

A number of other tool-oriented approaches have been described or developed that aim at extracting knowledge from game artefacts or properties in order to better understand the resulting gameplay (Nelson 2011). Shi and Crawfis, for example, presented a design tool that computes metrics on the optimal path a player may find to get through a level, given obstacles and enemy distribution (Shi and Crawfis 2013). They looked at the minimum damage cover, longest path and standard deviation of cover points. For RTS games, more general terrain metrics are also germane. Perkins' *Broodwar Terrain Analyzer* showed a map decomposition approach which allowed one to determine strategically important choke points (Perkins 2010), and Reddad and Verbrugge compute geometric centrality and coverage measures in order to better understand map quality (Reddad and Verbrugge 2012).

Tools that specifically facilitate design have also been developed. Liapis *et al.* presented a tool where designers could sketch levels using a high-level terrain editor (Liapis, Yannakakis, and Togelius 2013). Given a map, the tool outputs metrics such as playability, balance, choke points, *etc.;* the tool also included suggestions for map improvements based on a genetic algorithm approach that tries to produce playable maps. Tools that let the users enter a schematic for a level are presented as computational caricature tools by Smith and Mateas (Smith and Mateas 2011). Bauer *et al.* presented an extension of such caricature tool by letting it redesign the space of a level based on constraints given by the designer (Bauer, Cooper, and Zoran Popović). Their algorithm minimized an objective function subject to some set of constraints.

To the best of our knowledge no academic work has been done on specifically on sneaking AI in games. Outside of a game context, however, and particularly for military applications, stealth has been considered. Bortoff, for instance, describes an Unmanned Aerial Vehicle (UAV) design that avoids detection from radar (Bortoff 2000). His 2-step approach assumes non-moving enemies (radar), refining an initial path using a force-based solution expressed as a set of ODEs. Such continuous and real-world solutions are informative, although too computationally expensive for interactive use in games.

Tool Design

This section explores the workflow of the presented tool, the different technologies it uses, the different ways the user can visualize the results and how the interface is designed. The input is defined by a designed level that includes enemies with predetermined movements (routes), different geometries that block movement and/or enemy vision, and the player's initial position and goal. The tool then outputs a map with (clustered) possible paths a player could take to avoid detection.

We first describe the process that leads to this result. This requires discretization of the space, a pathfinding algorithm to compute sneaking paths, and a clustering approach to summarize results. This theoretical discussion is followed by description of interface itself, as an important part of the tool design.

Pre Computing - In order to find different paths a player could take we need a formal model of the game state. Enemy movements are assumed deterministic but possibly non-linear and non-continuous, and their vision is a function of obstacles and their orientation. Discretization of this space allows us to more easily model the game state as a function of time, and we thus compute the world state at different times, *t*, based on constant intervals. This gives us a 3D space to explore in order to find paths, as an extrusion of the 2D game level through time. Enemy vision is calculated using ray-cast methods (Vandevenne 2007), and this allows the tool to define enemy vision (FoV) at each point in time.

Rapidly Exploring Random Tree - We decided to use an *Rapidly exploring Random Tree* (RRT) for pathfinding through our discretized space, as it offers a flexible and inexpensive way to explore a state space. The random behaviour of the algorithm also allows the tool to easily represent a greater range of player behaviours. Here we describe the general motion planning problem for an RRT algorithm. Our game context does not require this full generality, but a general formulation enables further extension of our work.

The General RRT Algorithm - Initially presented by Kuffner and LaValle (LaValle and Kuffner, Jr 1999), RRT is an incremental, sampling-based, single-query motion planner for holonomic systems. The RRT has proved extremely useful in exploring configuration spaces and ultimately generating trajectories for robotic systems, with some recent applications in games (Bauer and Popović 2012).

In order to properly define the problem let $\chi \subseteq \mathbb{R}^d$ be the *state space* of dimensionality d where the elements of χ are known as *states* denoted as $\boldsymbol{x}$. The space occupied with obstacles is χ_{obs} leaving the rest of the space free χ_{free}. A *path* $\sigma \in \chi$ is a continuous function $\sigma(t) = (\boldsymbol{x}, t)$ connecting two states. It is said to be free iff $\sigma \in \chi_{free}$ and feasible if in addition it satisfies the system's kinematic/dynamic constraints.

Given an initial state $\boldsymbol{x}_{init}$ at t_0 and a goal region χ_{goal}, the motion planning problem deals with finding a *feasible, collision-free* path connecting the initial state to the goal region. The feasibility of the path σ is determined by not only residing in the free space but also governed by the dynamical/kinematic model of the system.

The RRT in algorithm 1 involves five main components and each can be encapsulated into a function or procedure as follows:

- *Problem Specifications* - A state space representation in which an initial state is used as the root node and finally a goal region. In this paper, we define the state space as $\chi = \{\boldsymbol{x} = (x, y, t); x, y \in \mathbb{R}; t > 0\}$. Since the time it takes to transverse a candidate trajectory is unknown the goal region is extended from a circle of radius r_{goal} centred at $[\mu_x, \mu_y]$ in $2D$ to an infinite cylinder. Formally defined as $\chi_{goal} = \left\{\boldsymbol{x} \in \chi; \sqrt{(x-\mu_x)^2 + (y-\mu_y)^2} < r_{goal}; t > 0\right\}$.
- *Sample* - When invoked, the sampling procedure returns a sampled state $\boldsymbol{x} \in \chi_{free}$ from a random distribution (uniform, normal with the goal as mean, or any other pseudorandom sequence as the designer chooses). A uniform sampling scheme in all dimensions is used in our work.
- *Steer* - Given two states $\boldsymbol{x}_1, \boldsymbol{x}_2 \in \chi_{free}$ the steer procedure attempts to connect these two states. Depending on the specific design, it either returns a boolean indicating connection success and/or a path segment $\sigma \in \chi$ such that $\sigma(0) = \boldsymbol{x}_1$ but not necessarily terminating at $\boldsymbol{x}_2$. In our case we use a simple straight line in the Euclidean sense for connecting the two states.
- *Collision Check* - Given a path $\sigma \in \chi$ or state $\boldsymbol{x} \in \chi$ this procedure returns a boolean true iff σ or $\boldsymbol{x}$ is within χ_{free} in its entirety; that is $\boldsymbol{x} \in \chi_{free}$. Essentially, the collision detection module is merely a "black-box" invoked by the algorithm. In our work, apart from checking collision with the obstacles (static and enemy FoVs) a kinematic constraint of maximal velocity is introduced and formulated as:

$$\begin{aligned} \textit{VelocityOK}(v) &= \begin{cases} \text{false} & : v \geq V_{max} \\ \text{true} & : \textit{else} \end{cases} \\ v(\boldsymbol{x}_1, \boldsymbol{x}_2) &= \frac{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}{t_2 - t_1} \end{aligned} \tag{1}$$

- *Metric* - A metric (or a pseudometric) is a function $\mathcal{D} : (\boldsymbol{x}_1, \boldsymbol{x}_2) \rightarrow \mathbb{R}^{\geq 0}$ that returns a scalar standing for the "distance" between the two states. For the sake of simplicity, we model the player with no dynamic constraints, although these can be easily incorporated as discussed in the conclusion. A simple straight line segment connecting two states suffices to represent a path segment along which the player can progress, giving us $\mathcal{D}$ as below. Note that only states with larger a time stamp than their parent states can be added to the tree (we require $t_2 > t_1$).

$$\mathcal{D}(\boldsymbol{x}_1, \boldsymbol{x}_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (t_2 - t_1)^2}$$

- *Nearest Neighbours* - Given a subset of states $\boldsymbol{V} \subseteq \chi$ and a state $\boldsymbol{x} \in \chi$, the *K nearest neighbours* (KNN) routine returns a subset $\boldsymbol{V}_{near} \subseteq \boldsymbol{V}$ of the nearest neighbours to $\boldsymbol{x}$ with respect to the metric $\mathcal{D}$. We implemented a k-d tree in order to improve performance of the nearest neighbours search (Berchtold et al. 2001).

In general, our implementation follows the algorithm 1 pseudo-code. The algorithm outputs solution paths $\sigma \in \chi$ connecting $\boldsymbol{x}_{init}$ to χ_{goal}. To shorten the time until a solution path is found, instead of biasing the random sampling distribution towards the goal, we attempt to connect each newly added state to the goal given it is within a predefined distance.

Algorithm 1 Path Finding

```
procedure COMPPATHS(x_init, χ_goal, χ_free, N, M)
Initialize(T, x_init, Σ)
    while i ≤ M do
        while j ≤ N do
            x_rand ← Sample(χ_free)
            x_near ← KNN(x_rand, T)
            [x_new, σ] ← Steer(x_near, x_rand)
            if CollisionFree(σ, χ_free) then
                T.edges ← T_edges ∪ σ
                T.vertices ← T_vertices ∪ x_new
                if σ.last ∈ χ_goal then
                    Σ ← σ
                    break
                end if
            end if
        end while
    end while
end procedure
```

The user controls the number of attempts the RRT can run before finding a solution, represented by N in algorithm 1. She can also specify how many times the RRT will run, M, which defines the upper bound on the number of paths the algorithm can find. The full set of paths output, Σ, are then clustered for presentation.

Path Clustering and Safe Spots - Showing all the paths on top of each other would have been cumbersome for the user, as there are many similar paths, and so it makes sense to use a clustering algorithm in this case. There are two clustering algorithms that we will discuss. The first one projects the dimension t onto one heat map plane, giving the user an overview of all the paths. The second takes into consideration t in the clustering, which means that the user can explore different paths in time by controlling t. Finally, it is possible for the user to view a map that identifies the "safe spots" for the player that are always free of detection.

2D Clustering - For any path σ in Σ they all leave a trace on the base 2D map, u. From there the map is normalized in order to clearly view the paths that have the highest value and to see the trace of the odd paths. This view gives a global vision of the paths and helps to see places where the user may not want the player to go. Figure 3 shows a specific example of this view. Although useful, a concern with this view is that it can lead to misunderstanding of the behaviour of the agents, since it is not obvious which paths are which after an intersection point.

3D Clustering - Sometimes the user would like to see all the paths at the same time evolved over time; this solves some of the 2D clustering problems. This is done by doing a cluster of all the paths on u at a particular t. By letting the user control the value of t used, she can see how the paths change over time. This allows for a closer analysis of what a player could accomplish in the level.

Safe Spots Finder - An interesting consequence of our

analysis is that we can easily show a summary of places where a player will never be seen by an enemy. For this we use the following mapping for $\boldsymbol{x} = (x, y, t)$:

$$colour(\boldsymbol{x}) = \begin{cases} \text{Red} & \text{if } !\chi_{free}(x, y, t) \\ \text{Magenta} & \text{if } \exists \tau \in T.\ !\chi_{free}(x, y, \tau) \\ \text{Green} & \text{if } \chi_{free}(x, y, \tau)\ \forall \tau \in T \end{cases}$$

An example is shown in figure 1. In this visualization it is clear which grid square the enemies will never see (shown in green). This gives the user a good overall understanding of the play space, allowing her to determine whether the safe places are there on purpose or not.

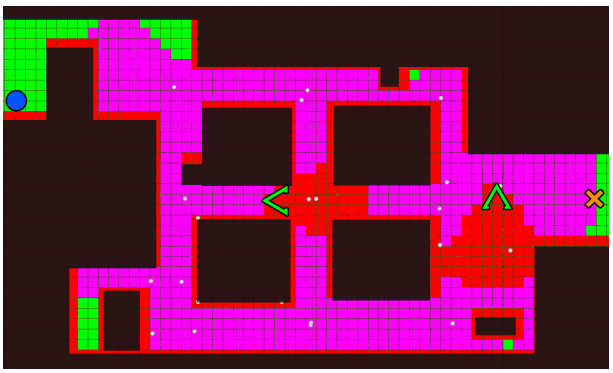

Figure 1: Game level plane in the *Unity 3D* environment. The player is represented as a blue circle, the goal as an orange cross and the enemies as green opened triangles oriented towards their field of view. Note the static obstacles in black, the safe spots in green, the spots where an enemy can see in magenta, and the enemy fields of view (and other unwalkable spots) in red. Small light dots are waypoints used to control enemy movements.

Interface - The mathematical goal of this work is to find paths that avoid enemies, but the main goal of the tool is to allow the game designer to easily understand and explore the search space. Parametrization of the tool is thus defined within *Unity 3D* as well. First, in the same panel used for final visualization (figure 1) the user specifies the enemy paths, fields of view, speed, *etc.*. Once the design of the level is finished, further initial parametrization is given through a separate control panel, outlined in green in figure 2. Here the user specifies parameters related to the pre-computation such as the grid size of the discretization and the time sample size. Basic initialization is completed by pressing the *precompute map* button at the bottom of the green panel.

Next, the user has to specify how the search is going to unfold; this is controlled through the panel outlined in red in figure 2. This input includes how many paths, *M*, the RRT should try, as well as the maximum number of attempts, *N*. After pressing *Compute Path* (bottom of the red panel), the search computation is performed and the visualization is constructed.

Once computed, the results can be explored in various ways, controlled by the panel outlined in blue in figure 2. The user can let the editor draw the last path found, the 3D

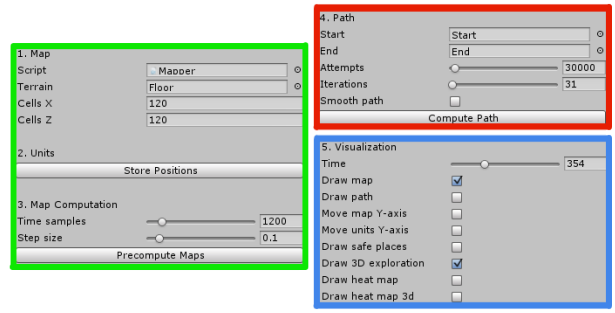

Figure 2: Tool interface panels in *Unity 3D*

search space, 2D or 3D heat maps, or safe spots, moving the visualization forward, backward, or to arbitrary points in time. This last step is the key interactive part of the tool for the user.

Experiments & Results

This sections explores tool behaviour. First we compare the paths found by the tool to those predicted ahead of time by a game designer. We then recreated the very first level of MGS to explore a fully designed space. We also used that same space to run performance analyses under varying parameters of our design.

Game Designer Level - During Smith's GDC presentation on stealth gameplay, he described a number of basic level designs for stealth. Figure 3 (a) (Smith 2006) shows one such situation, consisting of a corridor with a guard moving back and forth (red path). The expected player path is shown in blue, and Smith argues that players will wait in the alcoves to avoid the guard.

We redesigned this space in *Unity 3D*; figure 3 (b) shows the resulting space and heat map of possible player movements. As in Smith's design, the guard starts on the right (yellow dot), moving right to left and back (and repeating); the player starts on the left with the goal to exit on the right.

The heat map solution shows that the expected behaviour does not actually occur. While a majority of paths did head into the first south alcove and then to the north, a large number went directly to the north alcove. None of the paths actually entered the rightmost south alcove. Detailed exploration with our tool showed that once the guard has passed in front of the hidden player there is no need to hide anymore, and the player can proceed directly to the goal.

A designer might not have thought that it was possible to reach the north alcove without having previously reached the first south alcove, and so this may indicate weaknesses in the level design. In order to force the players to move into the first south alcove, the north alcove can be moved to the right, to the point where it is too far away to reach without being seen (figure 3 (c)). This achieves the goal of forcing the player into the first alcove, but that ends up obviating both the second and third alcoves. We experimented with a number of other variations in the level design and guard movement, and were only able to force the player to visit

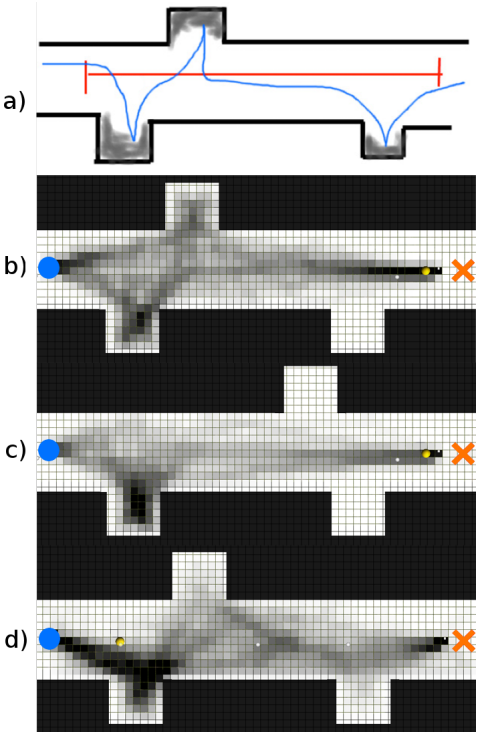

Figure 3: (a) Level design proposed by Smith (Smith 2006). (b) Heat map found from our tool. (c) Moving the north alcove. (d) Changing guard behaviour.

all 3 alcoves by modifying the guard's initial position and orientation, and having her stop and rotate/scan at multiple points in the patrol route (figure 3 (d)). Use of our tool to determine, correct, and validate the intended behaviour shows the value of such design exploration, resulting in a better level design and avoiding an expensive play testing cycle.[1]

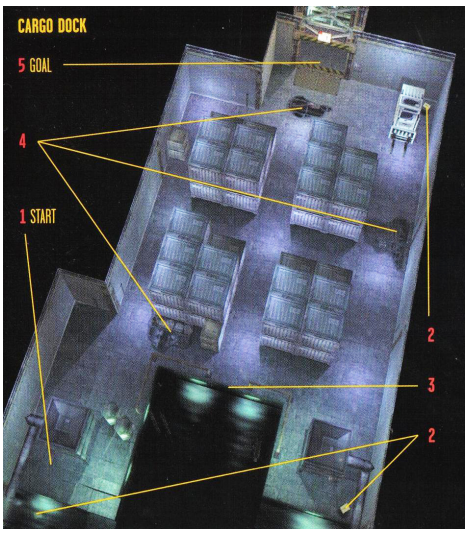

Figure 4: Metal Gear Solid's first level from the official strategy guide.

[1]For an interactive demonstration of the first part of this investigation see http://y2u.be/b8gSCG0Xs_k.

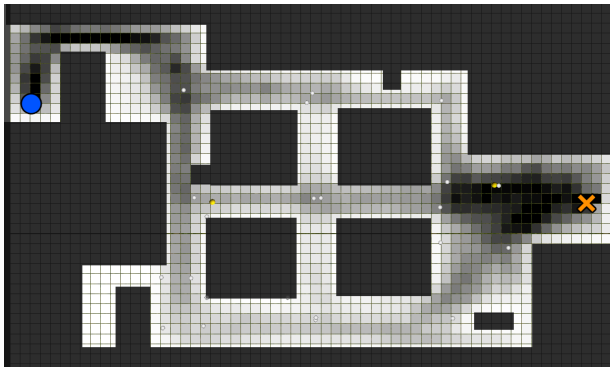

Figure 5: Metal Gear Solid's dock level's clustered paths found from the tool

Metal Gear Solid - This 1998 game is highly regarded as one of the first real stealth oriented games. We used the very first level of the game where the player is dropped in the enemy base's cargo dock and has to sneak around the enemies to get to the surface; see figure 4. In order to implement this level in *Unity 3D* we used a *sketchup* file found online (Anonymous 2007). This is a complex scenario with multiple path choices for the player. This is well represented by the thousands of paths found by the tool, again clustered and summarized as a heat map in figure 5. The two enemies (yellow dots) follow the same complex paths as in the game.

The results found show that the random search prefers the two top hallway over the bottom one. It is interesting to point out that no path was found to go in the lower left or lower right corners, although those locations do contain safe spots the player may aim for as intermediate goals (see figure 1); in the game they added objects for the player to collect at these points, making these spots more desirable. Note also that in the original game it was possible for the player to jump into the water (point 3 in figure 4) once seen and wait for enemies to lose interest. We cannot show this behaviour directly, as we focus on computing fully unseen paths, although it could be validated by ensuring no stealthy paths exist from the starting point, but do from at least one of the water exit points.

Performance Analysis - The interactive nature of our design hinges on the tool performing well for a reasonable range of parametrization. We thus explore the influence of three parameters over the time it takes to find a path and the probability of finding a path. We ran our test in the MGS level as it represents a real-life usage case. For every performance data point, we ran 150 iterations and for the probability tests every data point is represented by the number of tries the algorithm needed in order to find 150 good paths. Results were calculated on an *Intel* i5 at 3.00 GHz with 8 GB of RAM memory computer within *Unity 3D* 4.1.2f. While a given parameter is varying it uses fixed values for other parameters: 30 000 attempts, 1 200 time samples, and a grid size of 60×60.

Number of attempts - Varying the number of attempts the RRT does before stopping has a huge influence over the performance and the probability of finding a path. In figure 6 (upper left), one can see that the time taken to find a path in-

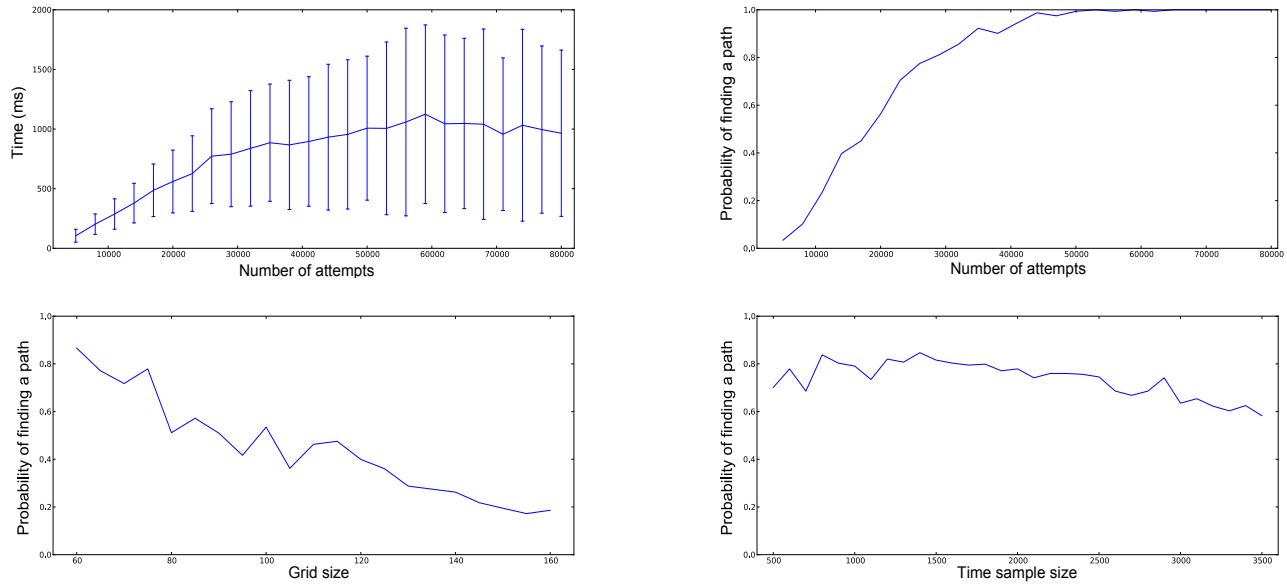

Figure 6: Performance analysis: attempts vs. time (top left), attempts vs. probability (top right), grid size vs. probability (bottom left), and time samples vs. probability (bottom right).

creases linearly with an increase in the number of attempts, and there is also a direct if noisy correlation with the probability of finding a path (see figure 6 (upper right)). This behaviour is mainly due to the fact that we used an uniform distribution over all search axes. Biasing the search toward the final or other heuristically determined goals would likely improve the performance significantly.

Grid size - This parameter represents a trade-off between granularity of the simulation and size of the search space. Fine-grain is naturally preferred, but as the space gets bigger picking a point within the goal region χ_{goal} gets less probable with the uniform distribution picking algorithm. This cost is clearly seen in figure 6 (lower left). Very reasonable grid sizes turned out to be effective in our experiments, so this is not necessarily a critical cost factor in practice, but using a grid is cumbersome, and although it greatly simplified the initial tool design, future development will explore use of a NavMesh within the full geometric space.

Time sample size - This defines the upper bound on the dimension t within the search space. The main impact of this parameter is on the probability of finding a path. If the value is too small the probability should decrease, as the need to complete the level quickly likely reduces the number of possible stealth paths. Our RRT approach performs quite well here, however, as shown in figure 6 (lower right). This suggests our tool may be also useful for finding out how fast a player may get through a level, and an appropriate visualization of such results is another part of our intended further development.

Discussion & Conclusion

In this paper we presented an AI-based tool to help game and level designers build and understand stealth levels. Our approach simplifies understanding of the many complex interactions that affect stealthy behaviour, and enables early detection of potential design concerns. The tool is integrated in the *Unity 3D* game development framework, and so provides a ready working space for exploration within an industrially relevant context.

There are a number of future directions we are exploring related to this work. Improvements to the RRT search are possible, for example, through use of biased search or by using a continuous space representation. We are currently exploring the influence of different game mechanics such as noise, light, walking *vs.* running, *etc.,* all of which are technically straightforward to incorporate, and would make this tool even more relevant to the reality of modern digital games. Moreover, representation is a key component of such a tool and we would like to explore feature-based state projections (Liu et al. 2011).

As pointed out in the introduction we are also interested in the possibility of applying this work to the problem of improving companion sneaking. This can be achieved via multiple approaches. We expect performance improvements to RRT have potential to enable online use, with some trade-off in complexity of implementation. Our early exploration has also shown that with sufficiently coarse clustering the number of feasible sneak paths may not be large, and thus pre-building a roadmap of stealthy routes for NPCs to exploit or recognize in player movements may also be possible.

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

Anonymous. 2007. Metal Gear Solid level 1 - Cargo Dock. http://sketchup.google.com/3dwarehouse/details?mid=29d5eae8d7a830261338ce2f5680446e.

Arkane Studios. 2012. Dishonored. http://www.dishonored.com/.

Bauer, A., and Popović, Z. 2012. RRT-based game level analysis, visualization, and visual refinement. In *Artificial Intelligence and Interactive Digital Entertainment Conference*, AIIDE 2012.

Bauer, A.; Cooper, S.; and Zoran Popović, title = Automated redesign of local playspace properties, b. . P. s. . F. y. . . p. . .

Berchtold, S.; Bhm, C.; Keim, D.; Krebs, F.; and Kriegel, H.-P. 2001. On optimizing nearest neighbor queries in high-dimensional data spaces. In *Proceedings of 8th International Conference on Database Theory*, ICDT 2001, 435–449.

Bethesda Game Studios. 2011. Skyrim. http://www.elderscrolls.com/.

Bortoff, S. 2000. Path planning for UAVs. In *Proceedings of the American Control Conference*, volume 1, 364–368.

Epic Games. 2008. UDK. http://www.unrealengine.com/udk/.

Konami Digital Entertainment. 1998. Metal Gear Solid. www.metalgearsolid.com.

LaValle, S. M., and Kuffner, Jr, J. J. 1999. Randomized kinodynamic planning. In *IEEE International Conference on Robotics and Automation*, volume 1, 473–479.

Liapis, A.; Yannakakis, G. N.; and Togelius, J. 2013. Sentient sketchbook: Computer-aided game level authoring. In *Proceedings of the 8th International Conference on Foundations of Digital Games*, FDG 2013, 213–220.

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

Looking Glass Studios. 1998. Thief: The Dark Project.

Naughty Dog. 2013. The Last of Us. http://thelastofus.com/.

Nelson, M. J., and Mateas, M. 2009. A requirements analysis for videogame design support tools. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, FDG 2009, 137–144.

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

Perkins, L. 2010. Terrain analysis in real-time strategy games: An integrated approach to choke point detection and region decomposition. In *Artificial Intelligence and Interactive Digital Entertainment Conference*, AIIDE 2010, 168–173.

Reddad, T., and Verbrugge, C. 2012. Geometric analysis of maps in real-time strategy games: Measuring map quality in a competitive setting. Technical Report GR@M-TR-2012-3, GR@M: Games Research At McGill, School of Computer Science, McGill University.

Shi, Y., and Crawfis, R. 2013. Optimal cover placement against static enemy positions. In *Proceedings of the 8th International Conference on Foundations of Digital Games*, FDG 2013, 109–116.

Smith, A. M., and Mateas, M. 2011. Computational caricatures: Probing the game design process with AI. In *Proceedings of the 2011 AIIDE Workshop on Artificial Intelligence in the Game Design Process*, IDP 2011.

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

Unity Technologies. 2013. Unity 3D. http://unity3d.com/.

Vandevenne, L. 2007. Lode's computer graphics tutorial - raycasting. http://lodev.org/cgtutor/raycasting.html.