AROB Practical Work

Hugo Mateo Trejo, Ángel Villanueva Agudo

***Abstract*— Practical work abstract. 200 words approx.**

I. Introduction

Navigation is a core component of modern videogames, as it determines how non-player characters move through complex environments while avoiding obstacles and respecting design constraints. It also lies at the heart of behaviour AI, since characters must not only reach their goals but do so in ways that appear intentional, legible and consistent with the game's fiction. From enemy flanking in action games to crowd movement in open worlds or unit coordination in strategy titles, robust navigation directly impacts believability and player immersion. At the same time, navigation systems must remain controllable and expressive enough to support artistic and level-design goals, so that designers and artists can obtain plausible, cinematic movement without manually scripting every trajectory.

Classical pipelines rely on graph-based representations (figure 1), such as navigation meshes or waypoint graphs, and search algorithms like A*, which require a carefully constructed graph discretizing the free space. In practice, these graphs are often built or curated by level designers, a tedious and error-prone process that is difficult to maintain in large or dynamic worlds. Automatic graph construction is possible but complex, hard to adapt in real time, and challenging to tune for artistic or gameplay requirements.

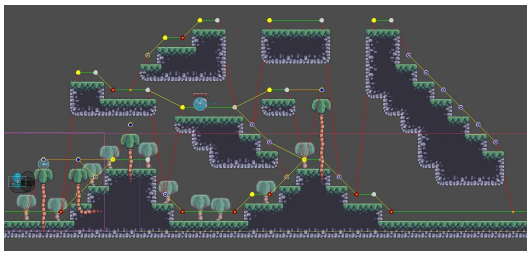

Fig. 1. Example of a graph navigation mesh in Godot Engine [1].

As an alternative, robotics research has developed plenty of methods that operate directly in continuous space, such as sampling-based planners and reactive navigation. However, many navigation pipelines designed for robotics are too computationally expensive to run in a videogame, since they are designed to work in a robot with its own dedicated resources. In contrast, videogame agents must share limited CPU time with other agents, rendering, physics, AI, and other systems, using frameworks that typically have very limited multithreading capabilities and need to run at a high framerate (60+ FPS). Thus, the challenge is to design a navigation stack that is both efficient and effective enough to run in real-time within these constraints.

This practical work implements a complete navigation pipeline based on these ideas: a customized RRT* algorithm acts as the global planner, computing collision-free paths in continuous space, while an efficient discretized holonomic ND controller serves as the local planner. Together, they provide a flexible, real-time navigation stack suitable for dynamic videogame scenarios.

A. Bob in the candyland

To showcase the navigation pipeline, a simple videogame has been created (figure 2). It consists on a 2D platformer where Bob the triceratops (the player) has to navigate through a procedurally generated candy maze to reach the end. The maze has bat enemies (agents) that pursue the player using the navigation pipeline described in this report.

To make the challenge more interesting the player has a special ability that destroys walls to create new paths in the maze, making it a dynamic environment.

In adition, it also includes a debug level to used in this report to test and visualize the navigation pipeline. It is fully implemented in GDscript (a python-like language), allowing easy deployment and modification, and can be downloaded here [Insert url].

Fig. 2. Main menu of Bob in the candyland.

B. Simulation framework

In Godot, the simulation runs in fixed *physics ticks* (usually 60 Hz), where the physics engine integrates bodies, resolves collisions and processes all `_physics_process()` callbacks. Navigation scripts run in this layer: at each tick they read agent states, call physics server queries (raycasts, shape-casts, area checks) to sense obstacles and path

visibility, and then write movement commands (e.g., target velocities) that the physics engine applies on the next step.

Mantaining this latency target consistently is crucial for responsive control: if navigation logic takes too long the physics engine will skip ticks to catch up, causing stuttering and unresponsive behaviour that completely break the user experience. Even more, repeatedly exceeding the budget may cause the physics engine to enter a "spiral of death" [2], a state where the engine fails to keep up with the target framerate, ticks accumulate infinitely (see figure 3) and the simulation indefinitely becomes unresponsive.

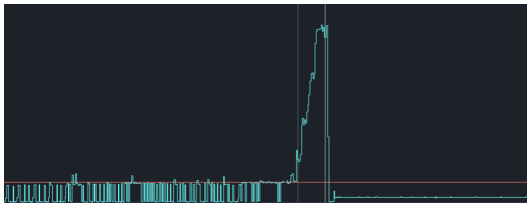

Fig. 3. Profiler visualization showing the "spiral of death" phenomenon, where physics ticks accumulate and cause the simulation to break [2]. The X axis is execution time, and the Y axis is physics latency. The red line is the target framerate (16ms).

Rendering and UI run separately in the variable-rate `_process()` loop and are not involved in the control logic, which keeps the behaviour deterministic and tied to the physics timestep.

C. The navigation pipeline

The system is built on top of the Godot game engine [3], which offers the physics and rendering capabilites needed for the simulation. The navigation pipeline consists on multiple components that interact with Godot to mimic the whole robotic navigation stack.

- **Global planner:** Computes a collision-free path from the agent's current position to a goal position using a custom implementation of the RRT* algorithm.
- **Trajectory follower:** This component is responsible for following the path generated by the global planner. It selects intermediate waypoints along the path and provides target directions to the local planner.
- **Local planner:** A discretized holonomic ND controller serves as the local planner. It takes the path generated by the global planner and computes movement commands to follow the path while avoiding dynamic obstacles.
- **Low-level controler:** It receives the commands from the local planner and executes them. Here the designer can implement movement mechanics like acceleration, jumping, etc.

II. Global planner: RRT*

The Rapidly-exploring Random Tree (RRT) [4] family provides efficient sampling-based planning in continuous, high-dimensional spaces. We employ a customized RRT* variant tailored for rapid replanning in dynamic environments.

The key design choice is goal-agnostic tree growth: each new sample is inserted by selecting the parent that minimizes cumulative cost from the root (the agent's current start), and only this root cost is stored. No heuristic or explicit goal-directed information is maintained during expansion, reducing bias and simplifying rewiring.

To obtain a path to any reachable target, the tree locates the best parent for that target and backtracks parent links up to the root, yielding the current minimum-cost route known currently. This allows to quickly extract paths to arbitrary goals without re-growing the tree, which is crucial to track dynamic targets.

During execution the agent's position diverges from the tree root, so when an unreachable path emerges or dynamic obstacles invalidate part of the current route, the tree needs to be rebuilt from the current agent's position.

A. RRT Display

To help with debugging and testing, a visualization system has been implemented to display the RRT* tree and the paths generated. It generates one visible point per tree node, and red lines that show the generated path by RRT*. These points and path display are updated every frame to reflect the current state of the tree and navigation.

Fig. 4. Display of the RRT* tree and path in the debug maze. Both are updated in real-time.

B. Dynamic obstacles

In dynamic environments, continuous sampling allows to naturally discover new paths that were not previously available. To allow doing that even with a filled tree, the planner allows to temporarily increase the tree size limit when a dynamic environment update is detected.

If the path becomes invalid due to a moving obstacle, the local planner will need to adjust or query a full global replan.

C. Collision checking

The physics simulation introduces some challenges that need to be addressed. To allow the algorithms to work with points and rays, the obstacles are usually enlarged by the agent's radius to prevent it from getting stuck at corners. However that is impossible to implement in the physics

engine, since it would require to manage two independant collision shapes per object that may change dynamically.

A work around could be to use Godot's *shape casting* feature, which allows to cast full shapes instead of rays. This would allow to check collisions of the agent's full shape along the edge, but comes with a significant performance cost in an already expensive algorithm. Instead, the paths are checked using simple ray-casts, and the final refinement is left to the local planner.

In this section, additional constraints could be added to only allow points that meet the kinematic constraints of the agent.

D. Real-time optimization

To ensure real-time performance, many optimizations and heuristics work together to speed up the location of valid paths and reduce the performance overhead of the tree maintenance. These optimizations tend to hinder the optimality of the solution, in favor of a more responsive and light behaviour in the most computationally demanding piece of the pipeline.

1) Sampling: These optimizations focus on reducing the sampling space and the number of samples needed to find a valid path, which improve responsiveness at the cost of path optimality and completeness.

- **Adaptive sampling:** Samples are generated uniformly within a disk around the agent and the current goal, focusing exploration on relevant areas. The size of the disk is configurable, and significantly affects the convergence speed. While key in the performance of the algorithm, this makes impossible to find paths path that go outside the disk, so more complex heuristics could be designed to adaptively increase the disk size when needed.
- **Sparse sampling:** When a sample is generated too close to an existing node, it is discarded and a new sample is generated. This prevents the tree from growing too dense in small areas, which would increase computation time without improving path quality. This threshold must be small enough to allow the tree to explore narrow passages, but even small values provide significant improvements in a space where the alternative is an infinitely dense tree.
- **Hot start:** When the agent visibility of the path is lost due to changes in the goal instead of dynamic obstacles, the previous path usually is still close to a valid one. Thus, the previous path is used as a seed to sample a percentage of the new points, which usually speeds up the location of a valid path significantly.
- **Tree size limit:** To reduce the tree-search space, there is a dynamic maximum point amount that depends on whether a path has been found or not. When there are no paths, the tree can grow larger to explore more space. When the map changes this amount is also increased temporarily to allow finding new paths. This again limits the ceiling of path complexity that cand be found, but it provides hard bounds to the computational cost of the planner which is crucial for a real-time system.

2) Rewiring: These optimizations focus on reducing the computational cost of the rewiring step, which is typically the most expensive part of RRT*.

- **K-nearest rewiring:** When a new node is added, only the k-nearest neighbors are considered for rewiring, limiting computational overhead to a constant amount. This directly limits the optimality of the paths, in comparison to considering all nodes within a radius. It is a direct trade-off between performance and path optimality. The bias introduced may be mitigated by the sparse sampling optimization, since a less dense tree will favor wider (and thus better) rewirings.
- **Quick build:** Each time the tree is rebuilt, a large number of samples is generated to quickly find an initial path. Each new sample tests the goal, and the process stops as soon as a path is found. Then, subsequent refinements will use fewer samples to optimize the existing tree in a goal-agnostic way. This helps to reduce the frame-time budget used by the planner while still providing fast responsiveness of the agent's movement.
- **Efficient nearest neighbor search:** A k-d tree structure is used to allow fast nearest neighbor queries during tree expansion. It was adapted from [5] to work with Godot's Vector2 class, and optimized significantly.

Together, these optimizations allow the RRT* planner to operate within a tight per-frame budget, finding valid paths in a few seconds even in complex, dynamic environments.

III. Trajectory following

Once the global planner provides a path, the agent needs to follow it. For that, the trajectory follower iterates over all path segments to find the closest one to the agent, and selects its second endpoint as the target to reach.

To prevent the agent from getting stuck at the endpoints, if the next segment endpoint has no obstructions the agent skips to it directly.

[Figure of a trajectory, the selected segment and the target point]

IV. Local planner: Discretized nearness diagram

The local planner is responsible for directly going towards the target provided by the trajectory follower, while avoiding immediate obstacles that may not be considered by the global planner.

It implements a discretized Nearness Diagram (ND) controller that reacts to nearby obstacles, offering local adjustments that prevent collisions with corners and small obstacles

undetected by the global planner without triggering costly replanning. Thus, it needs to be as lightweight as possible while providing thorough detection of immediate obstacles.

Instead of maintaining a continuous angular clearance map, the implementation samples the 360° space around the agent with eight evenly spaced proximity rays (every 45°). Each ray reports whether its direction is blocked by geometry, and those that are not obstructed are considered *valleys* (free sectors). When a valley is being considered for navigation, a full shape-cast with the agent's shape (typically a circle) is performed along the ray to refine the clearance. This refinement ensures the agent does not choose a direction that is free for a point but invalid for its actual footprint.

Given the goal direction, valley selection then finds the closest free valley to that heading by performing a linear search over them. To prevent jitter when several valleys are alternately opening and closing (e.g., at doorway edges), the controller applies a *hysteresis* rule. It stores the previously chosen valley and only switches to a non-consecutive valley if the angular improvement toward the goal exceeds a threshold ($\approx 135°$). Immediate switching is allowed between consecutive valleys (including wrap-around between the first and last sectors), which preserves responsiveness while suppressing large, unstable oscillations.

Obstacle avoidance and goal pursuit are combined additively. If the direct line toward the target is occluded (ray or shape-cast collision), the controller computes a repulsion vector by summing per-ray wall avoidance forces: each colliding ray contributes an outward normal scaled by how deep the collision point lies inside the sensing radius. This produces a smooth fleeing force stronger near closer walls. The final steering direction is the normalized sum of the valley heading unit vector and the accumulated repulsion. The result is a smooth, goal-oriented motion that moves around immediate obstacles without abrupt changes in direction.

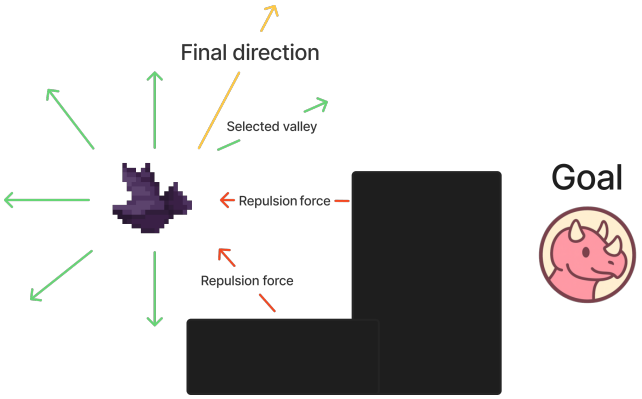

Fig. 5. Diagram with the main components of the discretized ND controller. 1) Proximity rays detect obstacles around the agent. 2) Free (green) valleys are identified and refined with shape-casts. 3) The selected valley is decided with hysteresis toward the goal direction. 4) Occupied (red) valleys contribute repulsion forces. 5) The best valley toward the goal is the combination of the selected valley direction and the repulsion vectors.

This reactive layer complements the global RRT* planner: the global module supplies a path in continuous space; the agent's controller decides which sub-point to follow, and the ND controller locally perturbs the motion to surround immediate obstacles without triggering full-tree rebuilds. By combining discretized angular sensing, hysteresis for stability, and additive wall repulsion, the approach yields smooth, goal-oriented motion suitable for small dynamic adjustments, while still remaining fairly inexpensive.

A. Performance considerations

Usually, physics queries are the most expensive operations in these algorithms. Thus, most effort is put into minimizing the number of queries performed each frame.

First, the agent has a circular area around itself that detects obstacles, permanently registered to the physics engine. This is an extremely efficient way to detect obstacles, but it only provides binary information (is there an obstacle inside or not). Once an obstacle is detected, the ND controller activates the 8 proximity rays (like a LIDAR) to get more detailed information about the obstacle's position. If there are no obstacles nearby, the ND controller is completely disabled to save computation time and the agent simply moves toward the goal.

Before performing the valley search, they rays are sorted by their angular distance to the goal direction. This favors testing the most promising valleys first, allowing to early-exit the search as soon as a valid valley is found. This optimization is particularly effective in cluttered scenarios where many valleys may be blocked, and directly reduces the number of shape-casts needed for clearance refinement. Ray-casts however are always performed for all 8 rays, since it is more performant to use Godot's dedicated raycast node.

V. Low-level controler

The low-level controler is responsible for executing the movement commands provided by the local planner. In this case, it simply moves the agent in the desired direction at a constant speed. This is the place where the designer can implement more complex movement mechanics like acceleration, jumping, etc, although many of those would require to modify the global and local planners to take them into account.

VI. Testing

A. Navigation testing

To test the navigation pipeline, a test maze (figure 6) has been hand-created to feature narrow passages with a complex path.

By just adding 10 new points per frame, RRT* + ND is able to find a path through the maze in an average of 3.7 seconds. While the path has not been found the local planner temporarily traverses the maze following any free passage, with the possibility of getting stuck in dead ends.

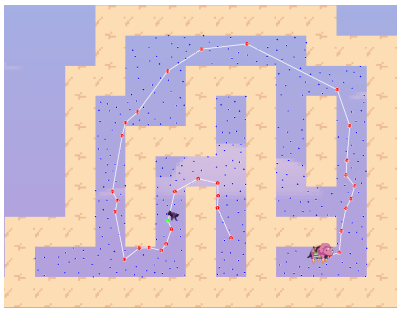

Fig. 6. Path generated in the demo level.

After opening a new dynamic path on the bottom of the maze (figure 7), the agent finds a new path in 0.3 seconds (which highly depends on the sampling to randomly find the new opened spot).

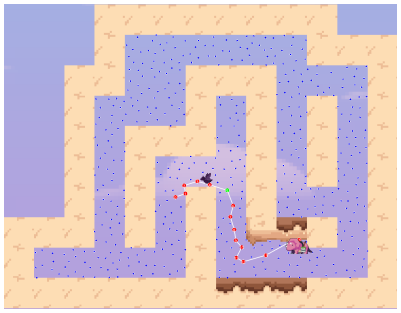

Fig. 7. Path generated in the demo level after opening a new path.

B. Very large path generation

In most short and medium cases, the path is found fast and nearly optially. The main problem this algorithm faces is when the required path is very large: first, the limited tree size makes it impossible to explore the whole space, although that could be reduced by dynamically increasing the tree size limit when the sampling disk is big at the cost of performance.

Second, the maximum sampling disk greatly improves performance and convergence, but it makes impossible to find paths that go outside of it. If the player is close to the agent but separated by a large obstacle, the agent will never be able to find a path to the player.

This is particularly noticeable in the hard level, which is a big maze (30x20 cells), and there are often enemies stuck because the path needs to go around large obstacles that are either outside the sampling disk or require a very large tree to explore the whole path.

This problem is inherent to purely sampling-based planners, and the only workarounds are using heuristics to increase the effective search area or using a more advanced planning algorithm.

C. Performance testing

Due to the large amount of optimizations, testing the impact of each optimization individually would take a lot of effort. Because of that the tests have been conducted using all of them aggregated, to evaluate the performance of the resulting system.

The navigation pipeline takes around 0.3ms per frame, mostly inside the get-k-nearest function. This is a heavily algorithmic function with a significant penalty coming from GDscript's interpreted nature.

Even with that, in the test scene completely remains stable even when new paths are opened dynamically, without significant latency spikes and always under 1ms of total script execution per frame.

D. Load testing

To test the navigation pipeline under load, a scene with 50 agents inside a procedurally generated 30x20 maze is used. The number of points generated per frame by RRT* is reduced to 3 to maintain performance, at the cost of increasing the time needed to find a path.

The scene runs smoothly at the standard 60 physics iterations per second, although the frame time (figure 8) has increased significantly to an average of under 10ms. This leaves still a good margin for other systems, which allows the game to run smoothly even with 50 agents navigating simultaneously with no significant freezes or stuttering.

Most of the time is spent again inside the get-k-nearest function, so a C++ implementation would significantly reduce this overhead.

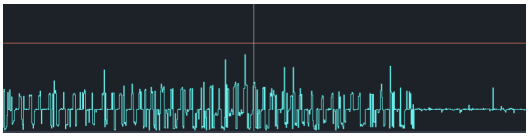

Fig. 8. Frame time of the execution with 50 agents navigating simultaneously. The red line marks 16ms, the budget for 60 ticks per second.

In any case, 50 agents are a lot for this kind of videogames, and videogames with significantly more agents (1000+) usually use much simpler systems due to other limitations.

Because of that the navigation pipeline is considered to perform well under load, with even more optimizations possible if needed.

VII. Conclusions

In this work we have implemented a complete navigation stack for 2D videogame agents, combining a goal-agnostic RRT* global planner, a trajectory follower, a discretized Nearness Diagram local controller, and a simple low-level movement layer. The system is fully integrated in Godot and drives non-player characters in a small platformer, where

agents must navigate a maze-like environment that can change over time as the player opens new passages.

The proposed RRT* variant grows a single tree in continuous space, supports rapid replanning, and can extract paths to arbitrary targets without rebuilding the structure from scratch. The trajectory follower turns these paths into short-horizon targets, while the ND controller refines motion locally using a small set of proximity rays, hysteresis for stability, and repulsion forces to slide around nearby obstacles. Together, these components keep the global planner lightweight, offloading fine collision avoidance and corner cases to the local layer.

Experiments in hand-crafted and procedurally generated mazes show that the system finds valid routes within a few seconds while staying within a tight per-frame budget, even with dozens of agents running simultaneously. The combination of sampling-based planning, discrete ND control and simple low-level actuation yields motion that is responsive, smooth and robust to dynamic changes, providing a usable navigation solution for videogame scenarios without sacrificing performance or designer control.

APPENDIX

A. Project structure

The navigation pipeline has two main folders:

- **Scripts/Actors:** Contains all the scripts related to the agents in the scene. Bat.gd contains the trajectory following logic and the low-level controller, which have been combined under the main agent controller due to their simplicity. Player.gd controls the player input and movement.
- **Scripts/Navigation:** Contains all the scripts related to the navigation pipeline. RRTStarNavigationGD.gd implements the global planner, HolonomicND.gd implements the local planner, and RRTtree.gd implements a helper KD-tree structure for efficient nearest neighbor search. All of them are combined inside Bat.gd, which queries a global path, computes a target from the trajectory, feeds it to the local planner and finally executes its movement commands and handles the animations.

The rest of the project contains an Art folder with the sprites used, World.gd which provides global convenience logic, Scenes which contains Godot scene files, and Scripts/Navigation/Display which contains the classes used to visualize the RRT* tree for debugging purposes.

REFERENCES

[1] Godot Engine, "Navigation in godot," https://docs.godotengine.org/en/stable/tutorials/navigation/navigation_introduction_2d.html.

[2] G. Fiedler, "Fix your timestep! an analysis on physics timesteps in games," https://gafferongames.com/post/fix_your_timestep/.

[3] Godot Engine, "Godot engine web page," https://godotengine.org/.

[4] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, Jun 2011.

[5] GeeksforGeeks, "Search and insertion in k-dimensional tree (kd-tree)," https://www.geeksforgeeks.org/dsa/search-and-insertion-in-k-dimensional-tree/.