AROB Practical Work

Author 1, Author 2

***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, 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.

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 demo level has been created to test and show the capabilities of the system, featuring moving obstacles and dynamic goals. It is fully implemented in GDscript (a python-like language), allowing easy deployment and modification, and can be downloaded here [Insert url].

A. The navigation pipeline

The system is built on top of the Godot game engine, 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) [Insert citation] 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, we locate the best parent for that target and backtrack parent links to the root, yielding the current minimum-cost route known by the current tree. 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 is rebuilt from the current agent's position. Continuous sampling naturally discovers detours around moving obstacles; while full rebuilds are triggered when the tree exceeds

a size threshold or connectivity becomes unreliable. This strategy preserves responsiveness without the overhead of maintaining a designer-built navigation graph.

A. 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.

B. Real-time optimization

To ensure real-time performance, several optimizations are implemented:

- **Efficient nearest neighbor search:** A k-d tree structure is used to allow fast nearest neighbor queries during tree expansion. It was adapted from [1] to work with Godot's Vector2 class.
- **K-nearest rewiring:** When a new node is added, only the k-nearest neighbors are considered for rewiring, limiting computational overhead to a fixed amount. However this directly limits the quality of the path found, in comparison to considering all nodes within a radius. It is a direct trade-off between performance and optimality.
- **Efficient 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, which directly affects the convergence speed. If the path goes outside the disk it will never be found, so the disk could by automatically increased if no path is found after some time.
- **Quick start:** 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, so 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.

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 and finds along them the minimum distance point to the agent. Having found the closest segment in the path, the target becomes the second endpoint of that segment. This is a generalization of the next waypoint in the path, which allows to compute it even when the agent is outside the path.

In the corner case where the agent is already at the computed target, the next segment is directly selected to prevent getting stuck at it.

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.

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. Performance analysis

VII. Functional testing

VIII. Conclusions

Practical work conclusions. Remember, it is part of your job to make us believe you deserve the maximum grade!!

Appendix

A. Project structure

The project has two main folders: Scripts/Actors and Scripts/Navigation.

- **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 RTTtree.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.

Player and World are configured as Autoloads, so they are automatically loaded as global Singletons when the game starts. Other bats can be added to the scene by just dragging Scenes/bat.tscn into the main scene.

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] GeeksforGeeks, "Search and insertion in k-dimensional tree (kd-tree)," https://www.geeksforgeeks.org/dsa/search-and-insertion-in-k-dimensional-tree/, accessed: 20 Nov 2025.