ABSTRACT

Title of Thesis: GENERATING FEASIBLE SPAWN LOCATIONS FOR AUTONOMOUS ROBOT SIMULATIONS IN COMPLEX ENVIRONMENTS

Rafael Florian Ropelato
Master of Science in Systems Engineering, 2022

Thesis Directed by: Professor Jeffrey W. Herrmann
Institute for Systems Research

Simulations have become one of the main methods in the development of autonomous robots. With the application of physical simulations that closely represent real-world environments, the behavior of a robot in a variety of situations can be tested in a more efficient manner than performing experiments in reality. With the implementation of ROS (Robot Operating System), the software of an autonomous system can be simulated separately without an existing robot. In order to simulate the physical environment surrounding the robot, a physics simulation has to be created through which the robot navigates and performs tasks. A commonly used platform for such simulations is Unity which provides a wide range of simulation capabilities as well as an interface for ROS.

In order to perform multi-agent simulations or simulations with varying initial locations for the robot, it is crucial to find unobstructed spawn locations to avoid undesirable situations with collisions upon start of the simulation. For this purpose, multiple methods were implemented with this research, in order to generate feasible spawn locations within complex environments. Each of the three applied methods generates a set of valid spawn positions, which can be used to design simulations with varying initial locations for the agents. To assess the performance and functionality of these approaches, the algorithms were applied to several environments varying in complexity and scale.

Overall, the implemented approaches performed very well in the applied environments, and generated mainly correctly classified locations which are suitable to spawn a robot. All approaches were tested for performance and compared in respect to their fitness to be applied to environments of varying complexity and scale. The resulting algorithms can be considered a efficient solutions to prepare simulations with multiple initial locations for robots and other test objects.

GENERATING FEASIBLE SPAWN LOCATIONS FOR AUTONOMOUS ROBOT SIMULATIONS IN COMPLEX ENVIRONMENTS

by

Rafael Florian Ropelato

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2022

Table of Contents

List of Tables

List of Figures

List of Abbreviations

2D	-	2-dimensional
3D	-	3-dimensional
ADAS	-	Advanced Driver Assistance System
CPU	-	Central Processing Unit
GPU	-	Graphics Processing Unit
HRI	-	Human-Robot Interaction
ISO	-	International Organization for Standardization
m	-	Meter
ms	-	Millisecond
RAM	-	Random-Access Memory
ROS	-	Robot Operating System
s	-	Second
UGV	-	Unmanned Ground Vehicle

Chapter 1: Introduction

Development of autonomous robotic systems is progressing rapidly. The official definition by the International Organization for Standardization (ISO) for a robot is: ”actuated mechanism programmable in two or more axes with a degree of autonomy, moving within its environment, to perform intended tasks” [1]. To achieve autonomy in the system, the robot needs to be able ”to perform intended tasks based on current state and sensing, without human intervention”. Autonomous mobile robots are already widely spread in industry as well as private sector where they manage warehouses or autonomously vacuum the living room. Other application for autonomous robots are surveillance or disaster response.

Autonomous robots require many sub-systems to work together to achieve the desired outcome. These activities include sensing the environment, planning motions and manipulation or actuation. Simulating the robots' behavior before deployment can help to assess the systems functionality. Using modern simulation environments allows the developers to test changes within the hardware and software without requiring costly alterations on the equipment or running time-consuming tests in a testing area. To perform such simulations, a physical model of the robot and the environment is required in which the implemented algorithms can be tested.

One of the most commonly used software platform for robots is the Robot Operating System (ROS). While ROS is not comparable to a conventional operating system, it allows all components of the robot to communicate and interact with each other. ROS provides a communication platform where micro-controllers can read data from sensors, perform motion-planning and send instructions to actuators to execute the desired motions. An enormous advantage of ROS is the ability to be installed on devices ranging from microcontrollers up to powerful processing systems. Using this capability, allows the developers to simulate robots on a Linux computer by feeding sensor data which can be gained from a simulated environment and observe the robots' behavior within the environment.

As ROS only provides a platform for the data exchange within the robot, this only covers the software side of the simulation. Over the past years, different platforms to perform the physical simulation have been used, including GAZEBO, V-REP or Unity. For this project, the Unity Game Engine [2] was chosen to design the environment and act as the physical part of the simulation. Unity is at its core a video game engine which includes powerful tools for visual rendering and uses the NVIDIA PhysX engine [3].

For simulating different situations that could be handled by autonomous robots, it might be of interest to run different scenarios with varying conditions. For developers, it is important to assess the robot's capabilities to handle a variety of scenarios with different initial conditions such as starting location of the robot itself or varying locations of objects which shall be reached or interacted with.

1.1 Aim of the research

The goal of my research project is to analyze approaches to generate different initial locations where a robot could be spawned in a complex environment. Such an algorithm could greatly reduce the time for developers to generate possible spawn locations for both the robot or goal objectives within a defined scenario. Using Unity's scene editor allows me to create different test environments which act as a foundation for the algorithms to find potential spawn locations.

All algorithms are tested in environments of different size and complexity to gain information about their performance. The collected data can then be evaluated to match the best suited methods for the different use-cases. The end result should give an overview about the expected processing time as well as the success rate of each algorithm depending on the environment in which they are deployed in.

1.2 Thesis Structure

To efficiently present the research I conducted,this thesis is structured according to the performed workflow. This section gives a short overview of the structure of my work and descriptions for each chapter and what it will include.

1. Introduction – The introduction chapter gives an overview of the background in the field, the aim of this research and the use of the work conducted.

2. Related Work – As the field of autonomous robotic simulations is an extremely active field of research, this chapter summarizes of the existing work related to my research.

3. Research Approach – This chapter discusses the approach to conduct my research and the tasks performed to implement and test the approaches.

4. Approaches – Here I will describe the different approaches to generate feasible spawn locations with description of the algorithms.

5. Experiment Model and Setup – The model of the robot as well as the different test environments and their application are described in this chapter.

6. Experiments and Results – This chapter includes descriptions of the conducted experiments and discussions of results.

7. Conclusion – Summary of this research and its outcome as well as limitations and potential future work.

Chapter 2: Related Work

2.1 Robot Operating System (ROS)

The complexity of robotic systems is increasing and requires a framework that allows onboard as well as off-board hardware to communicate in a standardized protocol. The most commonly used platform for robots is the Robotic Operating System (ROS) which was introduced in 2009 by Quigley et al. [4]

ROS is not an operating system in the conventional way, but a toolkit which allows different pieces of hardware to communicate in a clearly defined fashion. This framework uses topics to which one system can publish data or instructions, while other systems can subscribe to topics to receive the published information. This architecture allows all components of a robot, such as cameras, path-planner or actuators, to communicate even if they use different programming languages or interfaces. Using this standardized communication platform allows developers to reuse drivers and algorithms for different platforms and reduces the development efforts by minimizing the time required to rewrite code and drivers for a new system.

2.2 Unity Engine

By using the ROS package, which would be deployed on to the robot, the same packages can be used to simulate the system's behavior in a virtual environment. This approach saves time and reduces costs during the development, as there are little to no expenses to create a test environment or deploying a prototype of the robot. Using a simulation which closely represents real-world conditions can reveal flaws in the hardware or software of the robot and can be corrected in an early development stage. Furthermore, there is less effort required to make changes to the environment which allows the developers to test their set-up in a variety of situations with ease.

To represent the physical environment and behavior, there are numerous physics engines which can represent a model of the robot as well as its surroundings. The Unity Game Engine is one of the commonly used frameworks. Unity was designed as a game development platform which allows even inexperienced developers to create complex products. While the main focus of the engine is set on video games, it also includes all the desired functions for physical simulations. Because of its wide range of methods, it is frequently used for research involving 3D physics simulations and animation.

The platform called ”The Robot Engine” by Bartneck, et al. [5] utilized the Unity engine for work in the field of human-robot interaction (HRI). ”The Robot Engine” allows non-programmers to animate and control robots using a visual interface. This platform uses Unity as the main middle-ware which receives information from cameras and microphones, translates them into relative positions of a modeled robot, and finally sends the control commands to the robot which interacts with a person.

The Unity engine also provides a solid ground for AI research and training purposes. In ”Unity: A General Platform for Intelligent Agents” by Juliani et al., the authors argue that modern game engines such as Unity are suited as general platforms for the development of intelligent agent simulations, as they offer a rich complexity for visual and physical representations of real test environments [6]. Their paper focuses on the performance of Unity and the Machine-Learning-Agents Toolkit.

The work of Hussein, et al. [7] connects Unity with ROS to simulate the controls of autonomous cars in a traffic environment. The authors of the paper developed a 3D simulator for cooperative advanced driver assistance systems (ADAS) and autonomous robot by using the advantages of Unity and its 3D simulation capabilities with a ROS system architecture for autonomous vehicles. At the time when their paper was published, there was the issue that Unity was only available on Windows, while ROS was only working on Linux. Therefore, the authors had to implement a ”rosbridge” module which allowed simulations on both GAZEBO (Linux) and Unity (Windows) to work together and publish/subscribe to the same ROS topics. By now, Unity is also available for Ubuntu and other Linux distributions.

Using the Unity engine allows developers to create a physical simulation of the system’s environment with increased level of detail compared to other physics engines. With implemented optimization tools, the developed simulations can be designed physically and visually near-accurate to real-world test environments while still maintaining a moderate performance. According to the mentioned research papers, Unity is a good suit for autonomous robotics simulations and is also widely used in the research community.

2.2.1 Unity Simulation Design

The different environments in Unity are called scenes. In traditional video games, these scenes are levels or menus and build a canvas of the game, where all GameObjects are placed. The same concept applies for this research project. The scene includes the environment for the simulation as well as the model of the robot. For Unity, every object in the game is a GameObject. This includes elements such as the camera, terrain or structures like houses or trees. Each GameObject can be fit with different components that give the objects their desired behavior. Unity already comes with many prefabricated components like physical or rendering properties. Another component that can be added are scripts. For scripting, Unity supports natively the programming language C#. However, other .NET languages are also supported as long as they can be implemented as .dll files [8]. These scripts can be used to dynamically control the GameObjects and are the backbone of simulations or video games as they give objects the ability to interact and affect each other.

Unity offers options to create primitive shapes like boxes, cylinders or spheres as well as more complex tools to create terrain or water. Furthermore, it is possible to import 3D objects that were created with other software such as Blender [9] that allows the use of more complex geometries such as robots. Another advantage of Unity is the implemented Asset Store [10] which gives the developer access to an enormous library of assets which can be purchased or are free to use for the project. This allows developers to use complex models for the scene without the necessity to spend a large amount of time on creating detailed 3D models.

2.2.2 Unity Physics Engine

The NVIDIA PhysX engine, integrated in Unity, can be used to simulate realistic physical behavior such as gravity, impulse acting on an object or friction between objects. For my research, the most important aspect is the integrated collision detection. Using colliders and overlap or collision checks, it is possible to determine if any object is overlapping or touching another object in the scene. The Unity physics library also includes methods for distance measurements such as RayCasts, which casts a ray with defined origin and direction, to identify objects that are hit by the ray. Some of the features that may be used for my research are the following:

Colliders [11] - Using invisible boundaries around an object, called colliders, allows the detection of collisions between objects. These colliders are usually used to avoid objects to move through each other and allows them to exert an impulse force. Another use for colliders are trigger-events, which can be implemented to call a script or event to execute as soon as the collider of an object enters the collider of the trigger object. The simplest colliders are primitive colliders, such as box-, sphere- or capsule-colliders, which also require the least processing time. These primitive shapes can be added together and applied to a single game object to create a compound collider which can be used to better represent the object's actual shape. The more processor-intensive collider type are mesh colliders which use the objects mesh information to form a collider which represents the object more precisely.

RayCast [12] - Casting a ray which can be described like a ”laser beam”, which returns the colliders that are hit on its path. A ray-cast sends the ray from a given origin in a defined direction with an optional parameter to limit the maximum range of the ray. Different methods allow the ray-cast to return either the first collider that was hit or all colliders that were hit within the rays distance. More information can be returned form the raycast-hit such as distance or the point where the ray hit another collider.

Overlap - Using the overlap-functions, such as Physics.OverlapBox [13], allows the program to detect an overlap or enclosure of another object within the box whenever the method is called, even if the physics simulation is not running. Overlap functions are only available for primitive shapes like boxes, spheres and capsules. However, multiple overlap functions could also be combined to achieve a similar result like compound colliders.

While the frame rate of the simulation might vary during runtime, the physics computations are performed at fixed time steps. However, the physics engine is only running when the simulation is started. As for my research, there will be no real-time physics simulation be used as the spawn position should be generated before starting the simulation. An interesting solution to this issue was suggested by ”ThePilgrim” in the Unity answer forum, where the physics simulation can be advanced manually frame by frame [14]. Using this approach allows the evaluation of physics methods and collision detection without entering the runtime simulation.

2.3 Experiment Design for Data Acquisition

For reporting the acquired data and making claims about the result, the data should be presented following the guidelines of Jackson et al. [15]. The authors present three principles to follow when reporting computational experiments. Following these guidelines, the reported results are sufficient to justify the claims made and the information about the method and experiment setup is described in detail to allow reproducibility of the results. In their work, the authors do not create an exact set of rules to follow when presenting results, but lay the foundation to improve the quality for reports of computational experiments and reduce the effort of the readers to judge the results.

The authors Hall and Posner discuss the topic of generating experimental data for computational testing [16] and the principles for data generation. While this journal entry is based on their research for machine scheduling applications, the principles also apply for other computation testing research. The authors also base their key principles on the same guidelines as Jackson in and introduce the properties of experiments. These properties include the variety of problems, the practical relevance, the efficiency, and describability. As my work includes the generation of data and conducting computational experiments, both works are important and their principles are applied when conducting the experiments.

2.4 Conclusion

Research conducted in the field of autonomous robots and the application of 3D physics simulations mainly focuses on the robotic behavior and the solution to developing working algorithms to perform certain tasks. While there is a wide range of research concerning the implementation of ROS with adequate physics simulation tools, most applications rely on predetermined initial conditions for the simulation. For performing multiple simulations with varying initial conditions, or to generate simulations for multi-agent planning, it is crucial to be able to generate randomized initial positions for robots or potential target objects.

With the described implementation of ROS and Unity, these tools can be used to design versatile algorithms which can be applied for different test environments and would result in randomized initial conditions for robotic simulations. The common practice to run simulations with varying start conditions requires specifically designed environments or predefined areas in which an autonomous agent could start its task. With my research, I am planning to test different approaches and provide an overview of potential methods to generate random spawn locations within a complex environment.

Chapter 3: Research Approach

In my research thesis, I am testing approaches to find feasible spawn locations for objects, such as robots, in environments with different levels of complexity. Using such spawn positions gives researchers the opportunity to run simulations of robotic behavior with varying initial conditions. As the Unity engine offers an extensive set of functionalities, including an interface for ROS, it was chosen to serve as a basis for this research. Since Unity is also widely used in the video game industry, the result of this research could be used as ground work for a variety of aspects for game development or automation research.

With Unity offering a broad range of implemented functions, I first carried out a literature study to find existing approaches and applicable concepts to solve the issue at hand. As the task of generating spawn locations in a complex environment was assumed to be a common problem, I expected that there are existing solutions to the issue. However, the most commonly applied solution is to randomly pick and check a location in the environment in a trial and error procedure. This approach is less suitable for implementing ROS simulations, because ROS needs to be initiated using '.launch'-files which need to include the initial position for the simulation. Therefore, a predetermined set of locations from which a random entry can be chosen would be helpful to launch multiple robots or simulations in a given environment.

For my research, I developed and tested three different algorithms which yield lists of valid spawn-positions and pick a randomized set of entries from these lists. In the scope of this project, the focus was set to test algorithms which generate spawn points for autonomous ground robots only. The algorithms were applied in different environments which vary in scale and complexity. To run and test the implemented algorithms, a simple test environment was designed which includes a variety of obstacles and objects to test the accuracy and functionality of the code. As soon as the methods were functional, they were applied on large scale scenes which resembles a real-world example of an environment including buildings, lakes, and other obstacles.

All algorithms were tested with a set of input parameters to generate a benchmark for comparison. Testing the methods allows to determine their performance for a variety of use-cases such as different environment sizes or number of objects that shall be spawned. A clearly defined test approach was designed to generate data which can be used to compare various scenarios and applications for the different methods.

Chapter 4: Approaches

The approaches to find suitable initial positions for unmanned ground vehicle simulations are designed to evaluate locations for their validity to accommodate a robot without any collision. Therefore, all algorithms need to check the given locations for obstacles that might obstruct the spawn area. Additionally, the spawn locations should not be located too far above ground. As the algorithms were designed for ground vehicles, the robot simulation should start with only a slight drop. This chapter explains the concept of each algorithm and how it was designed to find suitable spawn locations as well as their implementation in Unity.

4.1 Feasibility Check

Each viable spawn location needs to satisfy a given set of conditions. In my case, I set the requirements for a spawn location to be feasible, if it does not touch another object and if the vertical distance to the terrain is below a certain threshold. Furthermore, the spawn location may not be located below a water surface. While the threshold for the height above ground can be easily adjusted as a value, the collision check needs to be implemented with colliders in both, the environment and the robot.

4.1.1 Collision Detection

In this research, a collision is defined as any two or more objects that either touch or overlap in the physical space. The algorithms assess a location's viability by checking if any object obstructs the space that the robot would occupy upon spawning. This can be achieved by mathematically comparing the areas which two objects occupy and check if these areas overlap. If the robot's collider area does not overlap with any other object, the location can be assumed to be collision-free.

4.1.2 Aerial Height Check

To assess the height above ground for in a potential spawn location, a simple height check can be performed. As the ground might not be a primitive plane but a terrain with a complex height map, the height can not just be determined by the vertical position in the coordinate system. Therefore a point-height-check can be performed using a vertical line segment between the center of the robot and the terrain. The length of the line segment yields the height above the point of the terrain that was intersected. This is a very simplified height check, as the robot has 4 potential contact points with the terrain, one for each wheel. Therefore, a situation might occur where the center of the robot can be considered above ground while one or more wheels are actually intersecting the terrain. Such a situation could be a spawn position near a steep slope. These exceptions are not considered in my research and may be implemented in future work.

4.1.3 Feasibility Check Procedure

With the established elements for the feasibility check, each selected coordinate can be assessed and is considered valid or invalid as a spawn location. Because the height check is mathematically less complex, it is possible to reduce processing time considerably by first checking for the height and if this check fails, the collision and water checks can be skipped. The following Algorithm 4.1 presents the basic steps taken to perform the feasibility check.

Algorithm 4.1: Feasibility-Check Algorithm

```
Data: Test Position ($Pos$), Height-Threshhold ($H_T$) and Robot Collider
Result: Viablity of Spawn Location
1  $height$ ← vertical distance from $Pos$ to Terrain;
2  if $height < H_T$ then
3      if $Pos$ below Water then
4          Return $false$;
5      else
6          move Robot to $Pos$;
7          if $robot\ collision$ then
8              Return $false$;
9          else
10             Return $true$;
11 else
12     Return $false$;
```

4.1.3.1 Unity Implementation

Implementing the feasibility check in Unity allowed me to check any predetermined location for its viability to spawn a robot. To perform the height check, a ray cast method was used. With a ray cast, it is possible to cast a ray similar to a laser beam, from a selected position, in this case the coordinates that shall be checked, and cast the ray in a selected direction. The ray can be defined to have a certain length which in this case can be the height threshold. When casting the ray, it returns all colliders that it intersected, as well as the distance at which the collider was hit. Using this information, the algorithm can detect if the ray hits the terrain or another object, such as a house. Only the terrain and objects on which the robot should be allowed to spawn are tagged with the tag "terrain". Therefore, it can also be examined if the ray intersects the terrain before any other object, which means that spawnable terrain is above any other object that might be covered in the scene. If a ray with the length of the height threshold intersects spawnable terrain as a first hit, the aerial height check returns $true$ and the collision check can be performed.

To assess if the current position is located under water, another ray cast can be performed in the upward direction. As water surfaces in Unity are usually implemented as planes rather than 3-dimensional water areas, a ray cast is used instead of a collision check. Performing a ray cast in the upward direction gives information if there is any water surface above the potential spawn location. If the ray hits a water plane, the examined position is marked as invalid.

Using the Unity physics engine, the collision detection is a simple method to check if two colliders intersect. For this, all objects that are considered obstacles, terrain, or water in the environment need to be fitted with a collider. Also, the robot model needs to have a collider component which is set to trigger mode. With the trigger mode, the robot does not physically interact with other objects and does not exert force on colliding objects or itself but a set of methods can be called, such as OnTriggerEnter() or OnTriggerExit(). As the spawn detection is happening before starting the robot simulation, therefore in edit mode, the physics engine is not running. Using a step-by-step simulation allowed me to advance the physics simulation manually one step at a time. Using this approach to perform a collision check, the algorithm could move the robot to the location that should be assessed, then advance the physics simulation and call the OnTriggerEnter() and OnTriggerExit() commands. Depending on the event of entering or leaving the trigger, the collision check can then return a $true$ or $false$ value and the tested position can be marked as a valid or invalid spawn point.

4.2 Methods

4.2.1 Random Search Approach

One approach that is commonly used in the video game industry, is to randomly search for a viable spawn position. With this method, a random coordinate within a predetermined area is selected and assessed for its viability. If the feasibility check fails, a new random location is selected and the process is repeated. As soon as a viable location is found, the coordinates are returned as a valid spawn location. To reduce computation power, already tested positions are stored and are not checked again. An additional parameter is the minimum distance to already tested locations. Using this parameter will make sure, that already assessed positions are separated by a minimum distance.

To avoid checking the same position multiple times, all previously tested coordinates are stored in a sorted list. Because each coordinate contains three dimensions, the list is be sorted by the X-coordinate. Searching a list of n elements results in a time complexity of $O(n)$ if the list is not sorted. When using a sorted list, a binary search can be conducted which reduces the search complexity to $O(log(n))$. As a certain minimum distance shall be maintained, all X-coordinates that are in the range of $\pm$ minimum distance around the currently tested coordinate. After the already tested coordinates that match this condition are selected, the euclidean distance between the assessed position and the already checked positions can be calculated and compared to the minimum distance. The following Algorithm 4.2 presents a simplified approach to find check the distance between already found locations and the currently assessed position. If none of the previously assessed positions are within the minimum distance of the current point and the minimum distance check succeeds, the new location is being tested and the feasibility check 4.1 can be performed.

Algorithm 4.2: Check Minimum Distance

```
Data: Position to Check ($P_{test}$), Minimum Distance ($dist_{min}$),
      List of checked Positions ($checkList$)
Result: Point too close to existing point ($true$ or $false$)
1 $limit_{left}$ = binarySearch($checkList.x$, $P_{test}.x - dist_{min}$ );
2 $limit_{right}$ = binarySearch($checkList.x$, $P_{test}.x + dist_{min}$ );
3 foreach $existingPosition$ in $checkList[limit_{left} : limit_{right}]$ do
4     if euclideanDistance($existingPosition$, $P_{test}$) $< dist_{min}$ then
5         return $true$;
6 return $false$;
```

4.2.1.1 Unity Implementation

To generate arbitrary test coordinates, a simple random number generator can be used to generate X-, Y- and Z-coordinates. Before generating those numbers, the user can determine a range in all three dimensions which allows to only check locations that are inside the environment or to narrow down the locations where spawn positions shall be generated. Once a position is generated, the new position is compared to all previously examined coordinates. If the new coordinate is not within the minimum distance to other positions, the previously described feasibility check is conducted and the returned value is used to determine whether the position can serve as a spawn location. This process is repeated until a feasible location is found. The following Algorithm 4.3 describes the processes for the random search.

```
Algorithm 4.3: Random Search Algorithm
  Data: Number of Locations to find ($N$), Height Threshhold ($H_T$),
        Minimum Distance ($dist_{min}$), Boundaries ($x_{max,min}, y_{max,min}, z_{max,min}$)
        Robot Collider
  Result: Set of Random Viable Spawn Locations
1 $P_{test}$ ;                                         /* Position to test */
2 $checkList \leftarrow []$ ;                /* List of checked positions */
3 $resultList \leftarrow [\,]$ ;         /* List of feasible spawn positions */
4 while $resultList.length() < N$ do
5     $P_{test} \leftarrow$ (Random.range($x_{max,min}, y_{max,min}, z_{max,min}$)); if
       checkMinimumDistance($P_{test}$ , $checkList$) == $false$ then
6         if feasibilityCheck($P_{test}$) == $true$ then
7             $resultList$.append($P_{test}$);
8     checkList.insertSorted($P_{test}$);
9 return $resultList$;
```

To performing robotic simulations with multiple robots, or to run multiple simulations with different initial conditions, a desired number of spawn locations (N) can be designated. Using this N, the algorithm can be repeated until N different and viable spawn locations are found. All assessed locations are stored to a text-file with the information if the coordinate is a valid or invalid spawn-point. Using this text file allows further processing to generate ROS launch-files.

4.2.2 3D Raster Search

Similar to the trial and error approach, the 3D raster search builds on the feasibility check method. To perform this raster search, the user can define boundaries which determine the three-dimensional range in which the search shall be conducted. A second parameter that needs to be defined is the step size which is used to generate a three-dimensional grid of coordinates within the defined range. Each generated coordinate is separated by the same distance defined as step size. Once the grid is created, the feasibility check is performed for every generated coordinate in the grid. Depending on the layout of the environment, it can be assumed that the majority of checked coordinates do not meet the aerial height constraint and can therefore be ruled out before the more complex collision check is performed. A simplified version of the implemented algorithm is shown in Algorithm 4.4 below.

Algorithm 4.4: 3D Raster Search Algorithm

```
Data: Height-Threshhold ($H_T$), Boundarys ($x_{max,min}, y_{max,min}, z_{max,min}$),
      Step Size ($Step$), Robot Collider
Result: Set of Viable Spawn Locations
$validSpawns \leftarrow [\,]$;
$invalidSpawns \leftarrow [\,]$;
for $x_{pos} = x_{min}; x_{pos} < x_{max}; x_{pos} + step$ do
    for $y_{pos} = y_{min}; y_{pos} < y_{max}; y_{pos} + step$ do
        for $z_{pos} = z_{min}; z_{pos} < z_{max}; z_{pos} + step$ do
            if feasibilityCheck($x_{pos}, y_{pos}, z_{pos}$) == $true$ then
                $validSpawns$.append($x_{pos}, y_{pos}, z_{pos}$);
            else
                $invalidSpawns$.append($x_{pos}, y_{pos}, z_{pos}$);
return $validSpawns$;
```

4.2.2.1 Unity Implementation

The implementation of the 3D raster search in Unity using the previously discussed feasibility check is straight forward. To check each position in the 3D grid, the algorithm can iterate through three nested for-loops which are limited by the user-defined boundaries in each dimension and the defined step size. Iterating through the loops, each position is checked by performing a feasibility check. The result of the check is then stored in a text-file including the locations coordinate and the state of the position as valid or invalid.

4.2.3 2D Raster Search

To reduce the number of iteration steps required to search the entire environment, a grid search in only two dimensions can be performed. This 2D raster search will generate a grid in the length and width dimensions. For each coordinate in the grid, the corresponding position projected on the terrain will be searched and evaluated. To project the 2D coordinate onto a complex terrain, a vertical line can be drawn that returns the X-, Y-, and Z-coordinates at which it intersects with the terrain. Using the exact location on the terrain surface, a user defined Y-offset can be applied to make sure the robot will not intersect with uneven terrain. At the generated test location, the defined feasibility-check can be performed and the result can be returned and stored. The following Algorithm 4.5 presents the steps to perform a 2D raster search.

4.2.3.1 Unity Implementation

The approach for implementing the 2D search is similar to the 3D raster search. The user defined boundary in the X and Z dimension and the determined step size are used to generate a 2D grid which builds the search raster. For each node in the grid, a ray cast can be performed that returns all intersection coordinates with the terrain. This also allows to check multiple positions that might be located above each other in a complex environment, such as positions on or under a

Algorithm 4.5: 2D Raster Search Algorithm

```
Data: Height Offset ($y_{offset}$), Boundaries ($x_{max,min}$, $z_{max,min}$),
      Step Size ($Step$), Robot Collider
Result: Set of Viable Spawn Locations
1  $validSpawns \leftarrow [\,]$;
2  $invalidSpawns \leftarrow [\,]$;
3  for $x_{pos} = x_{min}$; $x_{pos} < x_{max}$; $x_{pos} += Step$ do
4      for $z_{pos} = z_{min}$; $z_{pos} < z_{max}$; $z_{pos} + step$ do
5          ($x_{check}$, $y_{check}$, $z_{check}$) $\leftarrow$ projectOnTerrain($x_{pos}$, $z_{pos}$);
6          $y_{check} \leftarrow y_{check} + y_{offset}$;
7          if feasibilityCheck($x_{check}, y_{check}, z_{check}$) == $true$ then
8              $validSpawns$.append($x_{check}, y_{check}, z_{check}$);
9          else
10             $invalidSpawns$.append($x_{check}, y_{check}, z_{check}$);
11 return $validSpawns$;
```

bridge. Using the user-defined Y-offset and the found intersection with the terrain, a new point to check can be generated at which the feasibility check can be performed. The result of the check will be stored in a file including coordinates and validity of the spawn location.

Chapter 5: Experiment Model and Setup

5.1 Robot Model

For my research, the unmanned ground vehicle in use is the Husky UGV by Clearpath Robotics [18]. The Husky is a medium-sized ground vehicle used for exploration and manipulation with selected attachments. As it is capable of navigating through challenging terrain, it is a well-suited model to be applied to the different environments of this project. To integrate this robot into a 3D environment, Clearpath provides a 3D model of the Husky to be downloaded from their homepage.

The model of the Husky is imported to Unity and serves as a game object that can be equipped with physical components. To successfully perform collision detection and determine if a selected spawn location is unobstructed, the Husky needs to be fitted with a collider component. As the complexity of a collider could influence the processing time, especially for large numbers of collision checks, two different types of colliders were chosen. The most primitive and least processor intensive collider type is a box-collider. For this, only a bounding box will be applied to the 3D model which encloses the entire model in all dimensions. The more complex collider type which was selected is a convex mesh collider. While a mesh collider represents an object in its form more precisely, it is also more processor intensive and may therefore lead to increased processing time during collision checks. Unfortunately, the NVIDIA PhysX engine version 3.0

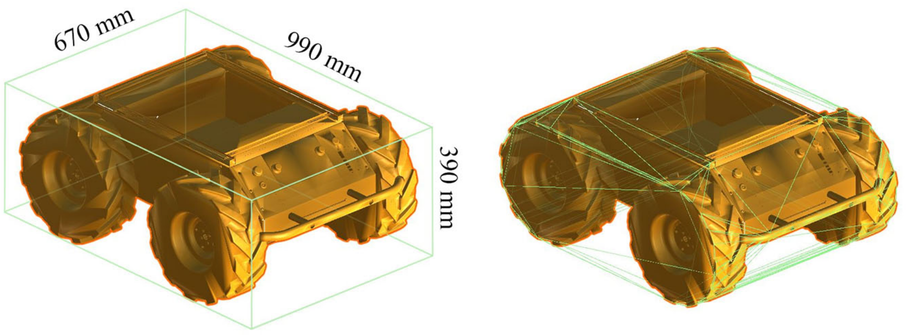


Figure 5.1: Husky with different attached colliders (green). Box-collider (left) and convex mesh-collider (right).

and above do not support concave mesh-colliders. The following Figure 5.1 provides an overview of the box- and mesh-collider applied to the Husky.

As shown in Figure 5.1, the mesh-collider represents the shape of the husky robot more detailed. However, this collider is also more complex and requires more computation time to check for collisions. In most cases, choosing a primitive collider is sufficient to achieve good results. Nevertheless, there are applications where a more complex mesh-collider is required. This representation of a convex mesh-collider uses the maximum possible number of 255 triangles that can be used for a single convex collider in the PhysX engine [17].

5.2 Environment

To test different methods to search for feasible spawn locations and assess their performance, I created different environments that vary in complexity. These scenes were designed to test the methods for different aspects, such as technical functionality, processing time, and application in complex environments. The following sections describes the various aspects of the different environments as well as their intended test applications.

5.2.1 Training Environment

In order to develop different approaches and being able to assess their functionality, a simple training environment was created. This scene resembles an arena with a level floor and four walls surrounding it. Inside the arena, there are numerous obstacles which are created from primitive objects like spheres and boxes. As this environment was mainly created to check if the algorithms are working properly, it is not a large-scale environment and only stretches over 20 meters in each direction. Figure 5.2 shows an aerial view of the environment as it was used to test the algorithms.

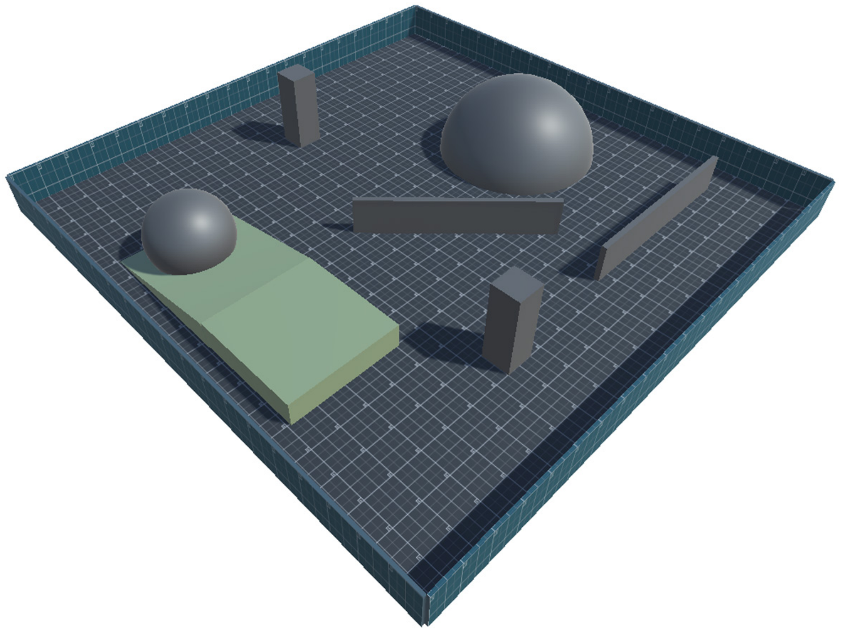


Figure 5.2: Aerial view of the ”Training Environment” created in Unity.

This training arena features a couple of obstacles such as walls, pillars or spheres and is enclosed in a set of four walls to each side. The level floor is tagged as ”terrain” and will be considered as a surface on which a robot can spawned. On the left side in Figure 5.2, there can be seen a ramp which leads to a platform which are both green shaded and are also tagged as ”terrain” for spawning. Therefore, the algorithm should only pick locations on the floor, the ramp or the platform as viable spawn points.

5.2.2 Low-Complexity Environment

As the training environment does not simulate real-world conditions, another scene was created which includes a forest area, a lake, a small town and an uneven terrain. This environment was used to assess the algorithms in a larger area which includes more complex structures. Since this environment is not highly complex, there are no exceptions for a robot to spawn but on the terrain. There are no open buildings or bridges included in this environment. Figure 5.3 provides an overview of the entire scene.

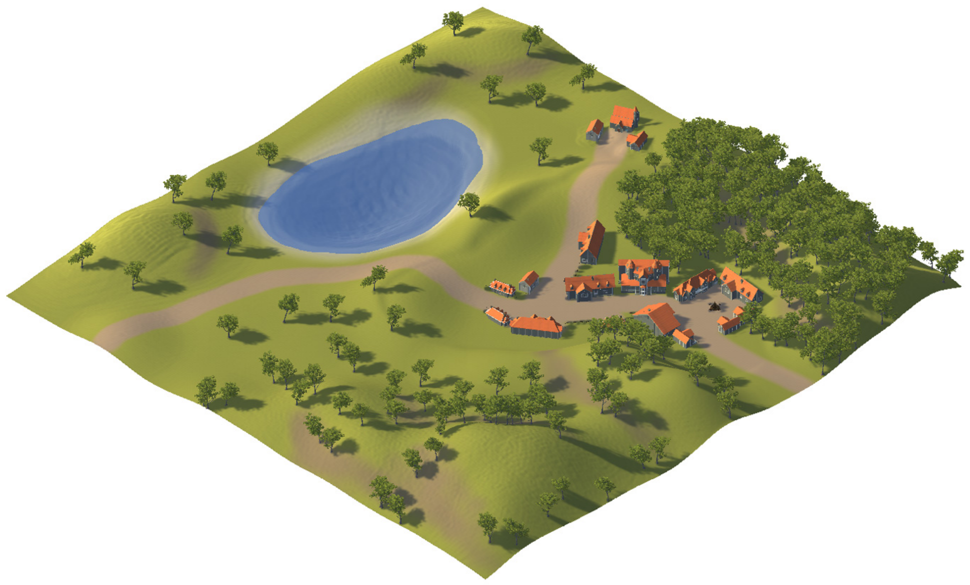


Figure 5.3: Aerial view of the ”Low-Complexity Environment” created in Unity.

The generated environment includes a small town, a dense forest on the right-hand side, a less dense forest with uneven terrain in the bottom area and a small lake to the left. It also includes a path which leads through the town. This path does not stand out from the rest of the terrain and is only a visual feature. The entire environment spreads over 500 meters in width and length and an elevation of about 75 meters measured from the ground of the lake. Figure 5.4 shows a close-up picture of the town square and gives a more detailed overview of the used houses, the fountain and benches. A list of all assets that I used in this work can be found in Appendix C.

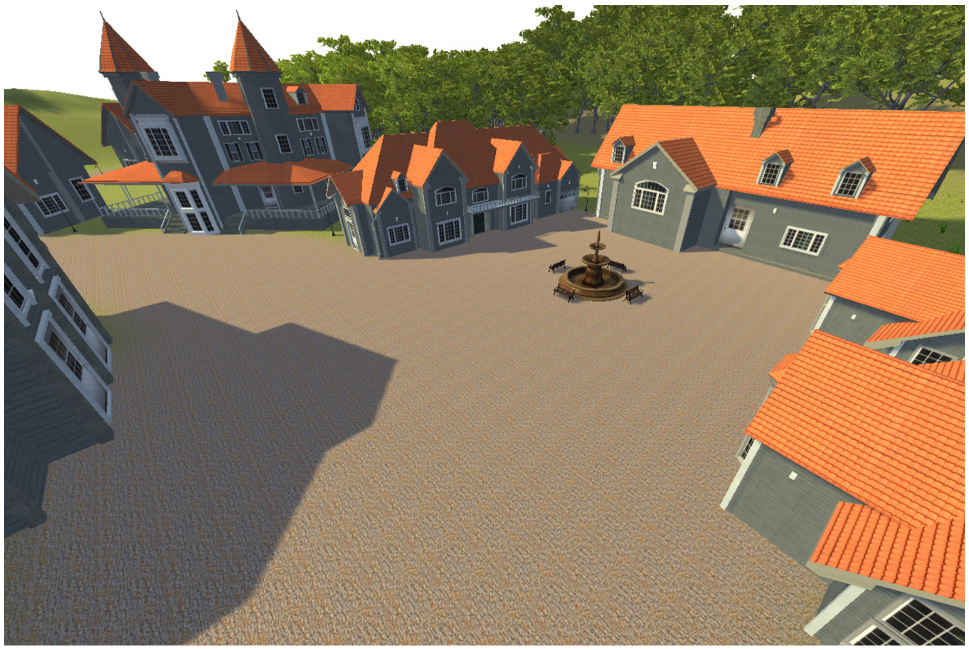


Figure 5.4: Overview of the town square and some of the used assets as obstacles.

This environment is not very detailed and does not exactly represent a real-world environment. However, it does allow me to test how the algorithms behave in different situations and can be used for benchmark tests in larger scenes. All objects in the environment were fit with compound box colliders or capsule colliders to serve as a base for the algorithm to work with.

5.2.3 Complex Environment

The most complex environment for this research represents a partially flooded area which can be used as a representation of a disaster area and simulations for recovery missions using autonomous ground robots. For my research, I used a prefabricated environment from the Unity Asset Store, the ”Flooded Grounds” environment. This scene includes highly detailed objects like houses, roads and fences, as well as uneven terrain and flooded areas. An additional feature,

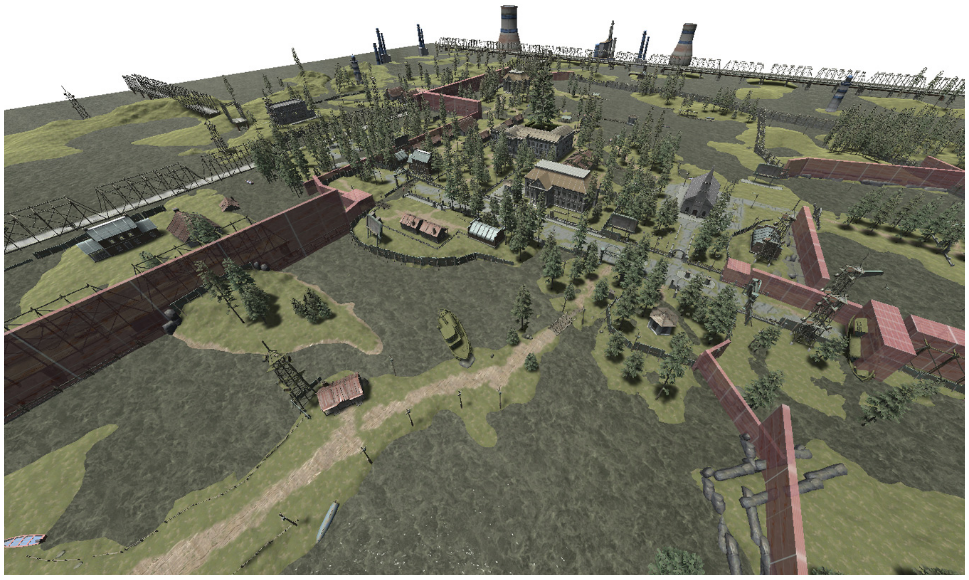


Figure 5.5: Aerial view of the complex environment, ”Flooded Grounds”.

which is not covered in the low-complexity environment, are imported 3D models that are marked as spawnable area. Therefore, the robot can not only spawn on the uneven terrain but also on bridges or roads. The Flooded Grounds scene also provides the largest environment for this study with a size of 1,300 meters by 1,300 meters. Figure 5.5 provides an overview of the Flooded Grounds environment.

As this environment was created as a setup for a post-apocalyptic video game, some changes on the environment were made. These changes include the removal of UFOs. Additionally, the trees in the scene were placed using Unitys terrain tool which allows placing randomized trees in the scene. Unfortunately, these trees do not interact with the collider as required for the algorithms to function correctly. Therefore, the trees were replaced with objects that include colliders.

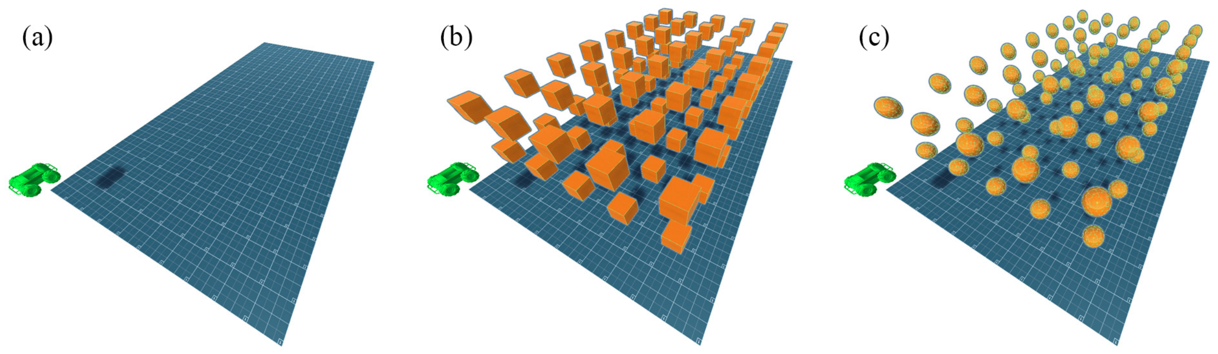


Figure 5.6: Overview of Benchmark Environment with different settings. (a) Empty Environment. (b) Populated with 80 box-colliders (5 x 2 x 8). (c) Populated with 80 complex mesh-colliders (5 x 2 x 8).

5.2.4 Benchmark Environment

To assess the computation time of different colliders, a benchmark environment was created. This scene can be filled with an adjustable number of obstacles. For this, an empty scene will be populated with two different versions of obstacles. For one, basic cubes with attached box-colliders will be used. As a second set of colliders, spheres with attached convex mesh-colliders with the maximum complexity of 255 triangles are used.

This environment can be used to run tests and determine the differences in computation time when using primitive or complex colliders on the robot as well as on the obstacles. The environment can also be chosen to be empty in which case only the aerial component of the feasibility-check is performed. Figure 5.6 provides an overview of examples for the three different set-ups. To perform benchmark tests, a larger quantity of objects can be generated. The plane only serves as a visual indicator of a floor and does not have a collider attached.

5.3 Environment Design

To support the developed algorithm, the environments need to be designed to clearly indicate what surfaces can be considered to be viable spawn locations. Making use of Unity's tag-system, each object in the Unity scene can be assigned to a specific tag. Using these tags, the feasibility check method determines whether an object within the maximum height constraint is considered terrain or an obstacle. Furthermore, certain environments need to be reworked and adjusted to include colliders for all objects that are to be considered an obstacle.

5.4 Experiment Design

The experiments to analyze the performance of each method are designed to measure their applicability for different use-cases. To analyze the performance of the fundamental feasibility check, a set of experiments will be designed to measure the algorithm's performance under clearly defined conditions. Using this approach, the processing time for the different aspects of the algorithm can be determined such as collisions or the aerial check.

The application for the 2D- and 3D search are very similar they will return all valid spawn locations that were found in the defined search raster. Therefore, both algorithm can be used to perform a full search of an environment and return sets of valid or invalid positions for a robot to spawn. As the use-case for both algorithms are very similar, their performance can be directly compared in terms of processing time depending on the density of the search grid. To assess the performance of the raster searches, both algorithms are applied to the low-complexity- and the complex environment. For each constellation, tests are performed using incremental step sizes and the results are compiled to be compared. The comparison of the full search algorithms can be used to determine the conditions for which either algorithm is best suited.

In order to assess the functionality of the search algorithms, a manual inspection is performed. For this, the generated array of valid and invalid locations are used to create a visual representation of the result. Each position will be represented with a marker in the scene view and be color coded depending on the classification as valid or invalid. This approach allows to manually observe the algorithm's decision making and check the position's classification for specific features in the environment.

With the random search algorithm, the outcome is a defined number of valid locations in the environment. As the processing time of this method is based on the probability to find a feasible spawn, this approach is considered nondeterministic. In order to generate a complete set of feasible spawn locations covering the entire environment, the application of this algorithm can result in excessive processing time. Therefore, the performance test for the random search algorithm is designed differently. The key metrics for this method are the average total processing time and the ability to find a set number of viable locations. As the algorithm shows a nondeterministic behavior, the average processing time and the variance of the result is calculated from multiple test runs. The algorithm will also be applied to the low-complexity- and the complex environment.

Chapter 6: Experiments and Results

In order to assess the performance of each method, the algorithms were applied to the created environments to run a set of experiments. The results obtained from the tests are compared to demonstrate the applicability and computation times depending on the complexity of the environment. All experiments were carried out on a personal computer with the following specifications. This chapter includes summaries of the test results. The detailed test results can be found in Appendix A.

Operating System	Windows 10 Pro
Unity Version	2020.3.24f1
Central Processing Unit (CPU)	AMD Ryzen 7 3700X
Physical Memory (RAM)	32.0 GB
Graphics Processing Unit (GPU)	NVIDIA GeForce RTX 2070-S

Table 6.1: Computer specifications for conducted experiments.

6.1 Benchmark Test

To investigate the processing time required for the feasibility check in environments with varying number and complexity of obstacles, the benchmark environment as described in Section 5.2.4 was introduced and will be applied to conduct benchmark tests. in the following section, the different experiment setups, including the independent aspect to be tested, are described.

6.1.1 Aerial Test

As the feasibility check examines the robot's height above ground in a first step, it is interesting to know how much processing time is used to perform a certain amount of aerial tests. This test includes the same code and steps that are used to perform the 3D raster search but all observed locations are considered to be in the air. Therefore, the same algorithm as for the 3D raster search is used but applied to an empty benchmark environment without terrain or obstacles.

6.1.1.1 Test Setup

To perform a large amount of aerial checks, tests with parameters as described in Table 6.2 were performed. As defined, the benchmark environment does not contain any obstacles or a terrain. Setting up the environment as described in the following Table 6.2 results in a behavior of the algorithm to only perform the aerial check for each feasibility check. The number of feasibility checks is chosen to be rather large even for test 1.1 as the processing time per aerial check is extremely low and large numbers of test yield a more meaningful result.

Test #	**Test 1.1**	**Test 1.2**	**Test 1.3**	**Test 1.4**
Nr. of Feasibility Checks	50,000	100,000	500,000	1,000,000
Test Raster (x, y, z)	(10, 10, 500)	(10, 10, 1000)	(10, 10, 5000)	(10, 10, 10000)
Step Size (x, y, z)	(1, 1, 1)			
Test Environment	Benchmark Environment			
Type of Obstacles	None			
Number of Obstacles:	0			
Test Algorithm	3D Raster Search			
Test Metric	Processing Time			
Observed Behavior	Processing time depending on number of aerial checks			

Table 6.2: Test 1 Setup for aerial check benchmark test.

6.1.1.2 Test Results

Table 6.3 presents the processing time results for the tests preformed as described in the previous Section 6.1.1.1. The results contain the total processing time to perform the designated number of aerial checks as well as the average time to perform 1,000 tests in the given environment. When taking a look at the time per 1,000 checks, it can be seen that the average processing time does not increase with the number of total checks. Also, the processing power required for this type of feasibility check can be considered extremely low compared to other feasibility checks.

Test #	**Test 1.1**	**Test 1.2**	**Test 1.3**	**Test 1.4**
Nr. of Feasibility-Checks	50,000	100,000	500,000	1,000,000
Total Processing Time [ms]	18.0 ms	34.0 ms	169.2 ms	342.0 ms
Processing Time per 1,000 Checks [ms]	**0.360 ms**	**0.340 ms**	**0.338 µs**	**0.342 ms**

Table 6.3: Test 1 results for aerial-check benchmark test.

6.1.2 Collision Test

While the aerial tests could be implemented in a straight forward fashion, the collision test does have more factors that can impact the processing time. The factors that shall be tested with this experiment are the complexity of collider objects. The benchmark environment will be prepared to include a number of obstacles and the robot for which the collision checks will be performed. The raster, in which the search algorithm will assess potential spawn locations, is aligned with the raster in which obstacles are spawned. Therefore, each feasibility check will result in a collision with an obstacle.

6.1.2.1 Test Setup

To analyze the influence of different collider models, the complexity of the robot as well as the complexity of the obstacles will be varied to run different test scenarios. For this, the robot will implemented with two different colliders, a box-collider and a convex mesh-collider. For the obstacles, the same variety of colliders is applied. This yields four basic test setups which will be set up as described in the following Table 6.4. An example for the setup of the collision test is shown in Figure 6.1.

Test #	**Test 2.1**	**Test 2.2**	**Test 2.3**	**Test 2.4**
Robot Collider	Box	Box	Mesh	Mesh
Obstacle Collider	Box	Mesh	Box	Mesh
Nr. of Obstacles	50,000			
Obstacle Raster (x, y, z)	(10, 10, 500)			
Search Raster (x, y, z)	(10, 10, 500)			
Test Environment	Benchmark Environment			
Test Algorithm	3D Raster Search			
Test Metric	Average Processing Time (10 repetitions)			
Observed Behavior	Processing time depending on complexity of robot and obstacle collider			

Table 6.4: Test 2 setup for collision benchmark test with different collider types.

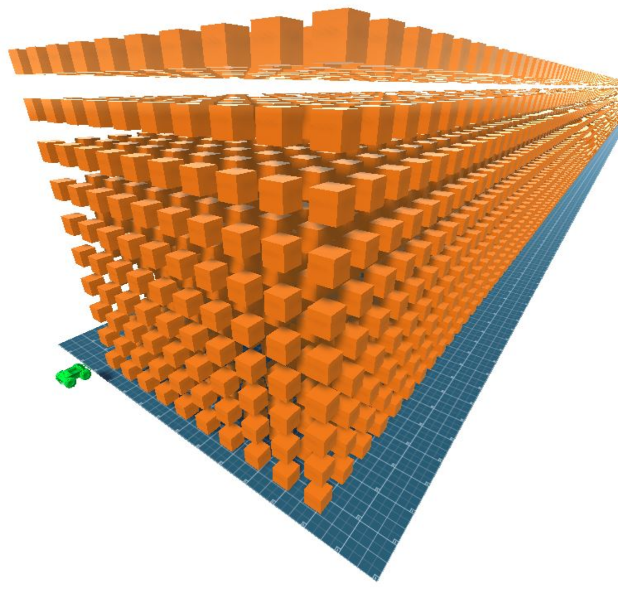


Figure 6.1: Overview of the collision experiment setup for test 2.1 and obstacles with primitive box colliders.

6.1.2.2 Test Results

All tests were executed as described in Table 6.4. To minimize the effect of background processes on the processing time, the total time was calculated from the average of 10 repetitions of each test. The results of the experiment are summarized in Table 6.5. The data provided by this experiment clearly shows longer processing times to perform collision checks compared to the aerial check. Therefore, the implementation of a preliminary check to rule out locations that are too far above ground greatly increases the performance of the feasibility check.

Test #	**Test 2.1**	**Test 2.2**	**Test 2.3**	**Test 2.4**
Nr. of Feasibility Checks	50,000	50,000	50,000	50,000
Nr. of Collision Detected	50,000	50,000	50,000	50,000
Robot Collider	Box	Box	Mesh	Mesh
Obstacle Collider	Box	Mesh	Box	Mesh
Average Total Processing Time [s]	10.63 s	10.89 s	10.18 s	11.15 s
Variance of Total Processing Time [s^2]	0.016 s^2	0.0081 s^2	0.2992 s^2	0.0397 s
Processing Time per 1,000 Checks [ms]	**212.6 ms**	**217.8 ms**	**203.6 ms**	**223.0 ms**

Table 6.5: Test 2 results for collision checks with different colliders.

While the complexity of the colliders would suggest that the computation time increases drastically when applying more complex colliders to the models, the data yielded by the benchmark test does not support this claim. Form the data gained by this test, it can be seen that Test 2.4, which incorporates mesh colliders for both the robot and the obstacles, results in the highest average processing time per check. However, there does not seem to be a difference or pattern in the resulting processing time between applying complex and primitive colliders.

6.2 Algorithm Application Testing

6.2.1 3D Raster Search

In this section, tests and results with the 3D raster search are described and analyzed. As the number of locations to be tested within a certain area increases depending on the step size, tests with a range of step sizes are performed and compared. The result of the applied tests give an overview of the processing time to perform a full search in 3 dimensions. In a further step, the intermediate results of the algorithm are examined by checking certain features of the environment and observing the decisions made by the algorithm.

6.2.1.1 Performance Tests

To assess the performance of the 3D raster search, the algorithm was applied to the low-complexity environment as well as the complex environment. For each scene, the algorithm parameters were adjusted to cover the entire range of the environment. The main impact on the processing time is expected to be the step size with which the search area is explored.

Low-Complexity Environment

For the first test, the search algorithm was applied to the low-complexity environment. The test was designed to perform searches with varying step sizes and gives an overview of the different processing times. Additionally, the ratio between valid, invalid, and aerial spawn locations can be compared between the different test runs. Table 6.6 provides an overview of the test setup. Figure 6.2 shows how the test area was selected to enclose the environment in its total height.

Test #	**Test 3.1**	**Test 3.2**	**Test 3.3**	**Test 3.4**
Step Size [m]	0.5 m	1 m	1.5 m	2 m
Nr. of Checks	≈ 187,400,000	≈ 23,600,000	≈ 7,000,000	≈ 3,000,000
Test Area (x, y, z)	From (0, 31, 0) to (500, 124, 500)			
Max. Height above Ground [m]	1.0 m			
Test Environment	Low-Complexity Environment			
Test Algorithm	3D Raster Search			
Test Metric	Processing Time, Ratio of valid/invalid Spawns			
Observed Behavior	Processing time and spawn detection ratio depending step size			

Table 6.6: Test 3 Setup for performance test of 3D raster search in low-complexity environment.

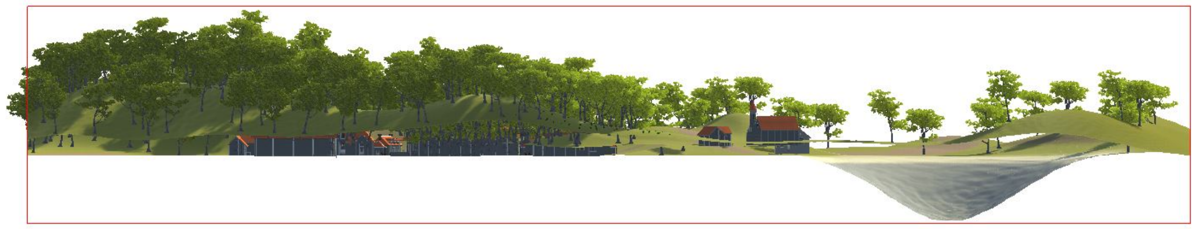


Figure 6.2: Overview of the chosen boundaries (red) for the 3D raster search applied to the low-complexity environment to include the environments total height.

The data gained from the performance test is shown in Table 6.7. The results clearly show, that the processing time increases exponentially with smaller step sizes. Another observation is the percentage of checked locations which are considered too far in the air or below terrain. With around 99.9 %, most of the checked locations already fail the aerial check. This shows that the implementation of this check reduces the processing time greatly when referring to the results in Table 6.3 and 6.5.

Test #	**Test 3.1**	**Test 3.2**	**Test 3.3**	**Test 3.4**
Step Size [m]	0.5 m	1 m	1.5 m	2 m
Nr. of Checks	≈ 187,400,000	≈ 23,600,000	≈ 7,000,000	≈ 3,000,000
Nr. of Valid Locations	≈ 1,730,000	≈ 217,000	≈ 68,600	≈ 30,800
Valid Locations [%]	0.926 %	0.923 %	0.976 %	1.04 %
Invalid Locations [%]	0.139 %	0.138 %	0.148 %	0.163 %
Aerial Locations [%]	98.93 %	98.94 %	98.88 %	98.80 %
Total Processing Time [s]	**363.4 s**	**47.4 s**	**14.0 s**	**6.6 s**
Time per 1000 checks [ms]	**1.94 ms**	**2.01 ms**	**1.99 ms**	**2.23 ms**
Ratio Invalid/Valid	**0.150**	**0.149**	**0.151**	**0.156**

Table 6.7: Test 3 results of performance test of 3D raster search in low-complexity environment.

Complex Environment

In a further test, the search algorithm was applied to the complex environment. The test was designed analog to the previous test for the low-complexity environment. However, as the scene is approximately six times larger than the previous one, the step size is adjusted to reduce the test duration. Table 6.8 provides an overview of the test setup. Figure 6.3 shows how the test area was selected to enclose the environment in its total height.

Test #	**Test 4.1**	**Test 4.2**	**Test 4.3**	**Test 4.4**
Step Size [m]	1 m	1.5 m	2 m	2.5 m
Nr. of Checks	≈ 132,800,000	≈ 39,400,000	≈ 16,800,000	≈ 8,700,000
Test Area (x, y, z)	From (-128, 0, -212) to (1152, 80, 1067)			
Max. Height above Ground [m]	1.0 m			
Test Environment	Complex Environment			
Test Algorithm	3D Raster Search			
Test Metric	Processing Time, Ratio of valid/invalid Spawns			
Observed Behavior	Processing time and spawn detection ratio depending step size			

Table 6.8: Test 4 Setup for performance test of 3D raster search in complex environment.

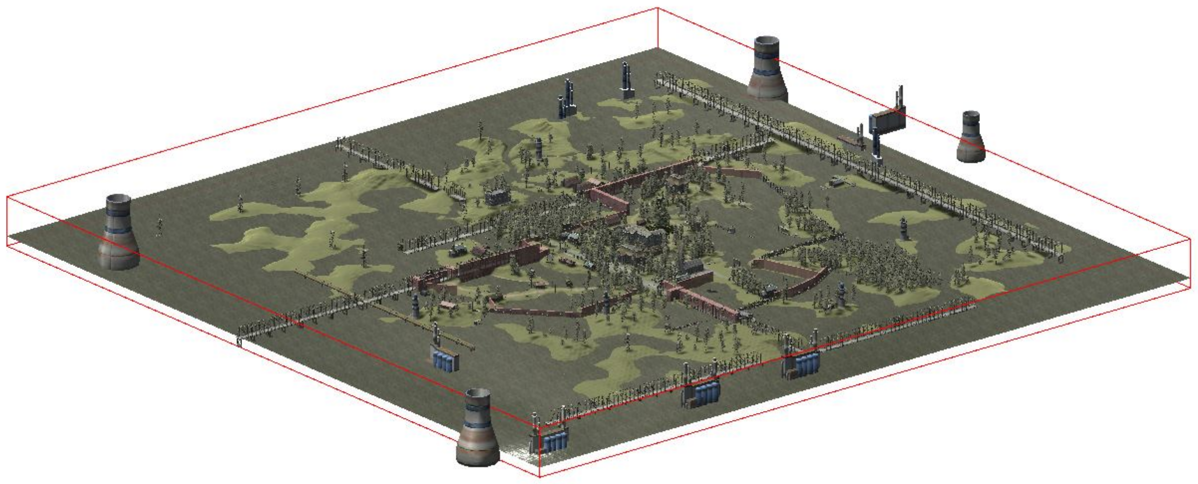


Figure 6.3: Overview of the chosen boundaries (red) for the 3D raster search applied to the complex scene to include the entire environment.

The data gained from this experiment is summarized in Table 6.9. The results look similar to those of the low-complexity test as they also show that most checked positions are located in the air. However, as this environment features a large water area, the percentage of invalid locations is larger than in the low-complexity environment. Furthermore, the percentage of valid locations in Test 4.2 is significantly lower than for the other tests. The reason for this occurrence is the line up of the raster with the terrain height. As the maximum height above ground is lower than the step size, it is possible that the terrain is at a height that is skipped by the raster search. As this environment is rather flat without any hills, some potential spawn locations might have been missed by the algorithm.

Test #	**Test 4.1**	**Test 4.2**	**Test 4.3**	**Test 4.4**
Step Size [m]	1 m	1.5 m	2 m	2.5 m
Nr. of Checks	$\approx$ 133,000,000	$\approx$ 39,400,000	$\approx$ 16,800,000	$\approx$ 8,700,000
Nr. of Valid Locations	$\approx$ 539,000	$\approx$ 125,000	$\approx$ 73,600	$\approx$ 40,100
Valid Locations [%]	0.406 %	0.317 %	0.437 %	0.462 %
Invalid Locations [%]	0.434 %	0.423 %	0.397 %	0.462 %
Aerial Locations [%]	99.16 %	99.14 %	99.01 %	99.88 %
Total Processing Time [s]	**161.5 s**	**41.8 s**	**20.6 s**	**10.9 s**
Time per 1000 checks [ms]	**1.22 ms**	**1.06 ms**	**1.22 ms**	**1.26 ms**
Ratio Invalid/Valid	**1.07**	**1.33**	**0.91**	**1.00**

Table 6.9: Test 4 results of performance test of 3D raster search in complex environment.

6.2.1.2 Functionality Tests

After examining the performance of the 3D raster search algorithm, I wanted to analyze the algorithm's functionality. For this purpose, I executed the algorithm to find valid and invalid spawn locations in both environments and present the result with markers in the scene view. This approach gives the opportunity to manually check if the algorithm determines the viability of a location as expected. Figure 6.4 shows a collection of special cases in the low-complexity environment and how the algorithm determines the validity of the locations. The examined locations are indicated with green markers for valid locations and red markers for invalid spawn locations.

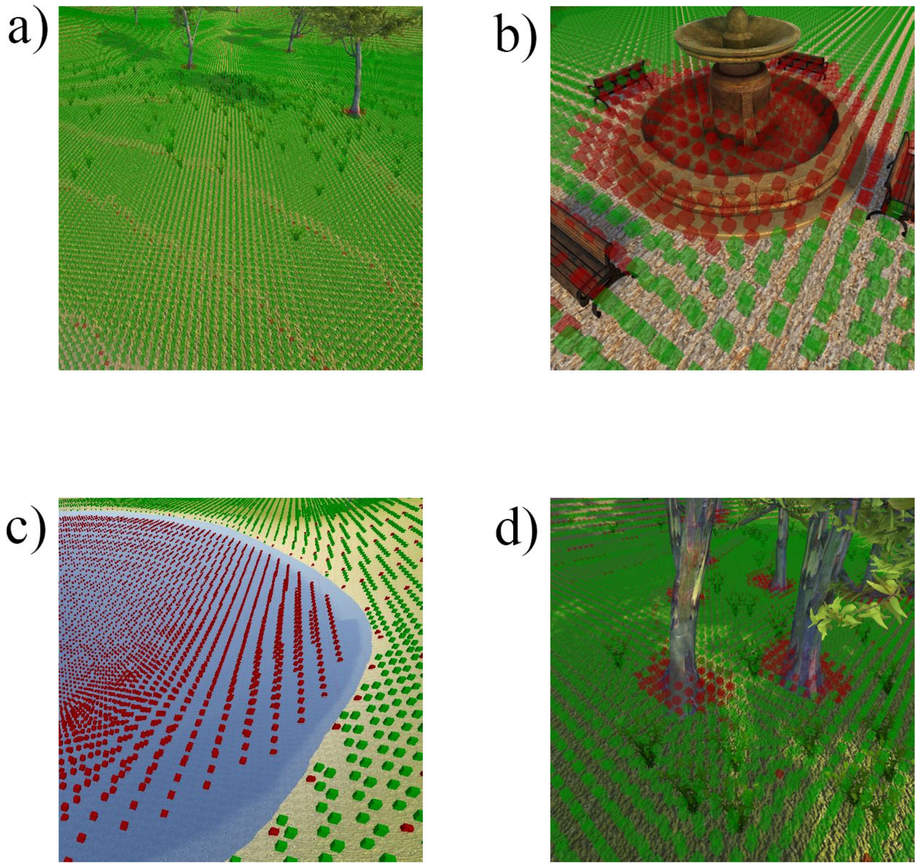


Figure 6.4: Overview of the classification of spawn locations by the 3D raster search algorithm in the low-complexity environment. a) Open terrain, mainly valid locations. b) Collision with fountain in town square. c) Detection of underwater locations. d) Collisions in dense forest.

As shown by the special cases in Figure 6.4, the decisions made by the search algorithm conform to the expected outcome. Generally, the open areas provide plenty of room for a robot to spawn. Therefore, the plains with little to no objects are mainly covered with valid spawn locations. In addition, all positions where the robot would collide with an object near trees, benches, or the fountain, are marked as invalid locations. Also, the algorithm detects areas which are in proximity to the terrain but underneath a water layer and stores them as invalid locations. This manual functionality check showed that the algorithm works well with the low-complexity environment.

In a next step, I applied the algorithm to the complex environment. As this environment was not specifically designed to accommodate this algorithm, the scene provides a good example for the application of the algorithm in a third-party scene. Figure 6.5 provides an overview of the algorithm's classification of spawn locations in different situations.

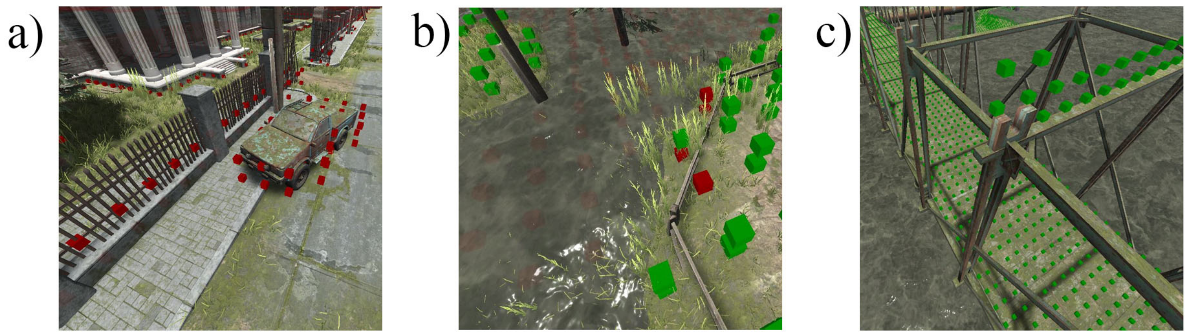


Figure 6.5: Overview of the classification of spawn locations by the 3D raster search algorithm in the complex environment. a) Collision detection in town. b) Water and collision detection with small objects. c) Spawn detection on bridge.

While the algorithm performs mostly correctly in this environment too, there are some decisions which could lead to an unsuccessful robot simulation. In example a), the algorithm correctly detects collisions with objects in the city. Therefore, locations which would be inside or too close to the abandoned car are marked as invalid. In situation b), the classification is also technically correct. Therefore, the locations in the water are marked as invalid and so are spawn locations where the robot would collide with the fence. However, some viable spawn locations are positioned above the fence. While the robot would not collide with the fence when spawned, it could lead to unwanted issues when the physics simulation is started and the robot might roll over after partially landing on the fence. As for example c), the spawn locations on the bridge are chosen correctly. However, as the entire object of the bridge is marked as spawnable terrain, this also includes the pillars and the frames of the bridge. Spawning a robot on top of the frames would lead to a failure of the simulation. However, this problem could be prevented by correctly editing the bridge object and clearly defining which part of the bridge should be considered to be valid spawn locations.

6.2.2 2D Raster Search

In a similar fashion to the tests performed with the 3D raster search, performance and functionality tests are conducted using the 2D raster search. Conducting performance tests with similar setup parameters as for the 3D search, the performance of the two algorithms can be compared in a later step.

6.2.2.1 Performance Tests

The 2D raster search algorithm was applied to both the low-complexity and the complex environment. For both scenes, raster searches with varying step size were conducted and the required processing time was analyzed. Again, the search parameters are designed to cover the entire environment.

Low-Complexity Environment

In a first step, the low-complexity environment is used to perform the first analysis. All of the following tests are performed with the same search parameters. The only variable that changes is the step size. I assessed the outcome of the processing time and the ratio of valid to invalid spawn locations. The search area was selected exactly the same as in test 3 and also covers the entire height of the environment as shown in Figure 6.2. Table 6.10 presents the setup for the test parameters.

Test #	**Test 5.1**	**Test 5.2**	**Test 5.3**	**Test 5.4**
Step Size [m]	0.5 m	1 m	1.5 m	2 m
Nr. of Checks	$\approx$ 1,000,000	$\approx$ 251,000	$\approx$ 112,000	$\approx$ 63,000
Test Area (x, z)	From (0, 0) to (500, 500)			
Ray Cast Height to Depth	124 to 31			
Chosen Y-Offset [m]	1.0 m			
Test Environment	Low-Complexity Environment			
Test Algorithm	2D Raster Search			
Test Metric	Processing Time, Ratio of valid/invalid Spawns			
Observed Behavior	Processing time and spawn detection ratio depending step size			

Table 6.10: Test 5 Setup for performance test of 2D raster search in low-complex environment.

Table 6.11 summarizes the results of the performance test using the 2D raster search on the low-complexity environment. The data shows, that the percentage of valid and invalid locations do not necessarily add up to 100%. As this algorithm only performs a feasibility check once the terrain was hit, there are locations along the outskirt of the environment which miss the terrain and therefore don't return any position, valid or invalid.

Similar to the 3D search, the processing time also increases with smaller step sizes. However, it can be seen that the processing time does not increase as drastically as in the equivalent test using the 3D raster search. This is mainly caused because the number of checks grows in a quadratic order, rather than a cubic one. The ratios of invalid to valid locations are in a similar scale as in the results of test 3.

Test #	**Test 5.1**	**Test 5.2**	**Test 5.3**	**Test 5.4**
Step Size [m]	0.5 m	1 m	1.5 m	2 m
Nr. of Checks	$\approx$ 1,000,000	$\approx$ 251,000	$\approx$ 112,000	$\approx$ 63,000
Nr. of Valid Locations	$\approx$ 896,000	$\approx$ 224,000	$\approx$ 100,000	$\approx$ 56,000
Valid Locations [%]	89.4 %	89.2 %	89.6 %	88.9 %
Invalid Locations [%]	10.4 %	10.4 %	10.4 %	10.3 %
Total Processing Time [s]	**114.7 s**	**28.6 s**	**12.6 s**	**7.2 s**
Time per 1000 checks [ms]	**114.44 ms**	**113.93 ms**	**113.37 ms**	**114.51 ms**
Ratio Invalid/Valid	**0.116**	**0.116**	**0.116**	**0.117**

Table 6.11: Test 5 results of performance test of 2D raster search in low-complexity environment.

Complex Environment

The same analysis was then applied to the complex environment. The setup for the test is identical with the setup in test 5. However, the step size is adjusted to match the step sizes in test 4. I again observed the resulting processing time and the ratio of valid to invalid spawn locations. The search area was also designed exactly the same as in test 4 to cover the entire environment as shown in Figure 6.3. Table 6.12 presents the setup for the test parameters.

Test #	**Test 6.1**	**Test 6.2**	**Test 6.3**	**Test 6.4**
Step Size [m]	1 m	1.5 m	2 m	2.5 m
Nr. of Checks	$\approx$ 1,640,000	$\approx$ 729,000	$\approx$ 411,000	$\approx$ 263,000
Test Area (x, z)	From (-128, -212) to (1152, 1067)			
Ray Cast Height to Depth	0 to 80			
Chosen Y-Offset [m]	1.0 m			
Test Environment	Complex Environment			
Test Algorithm	2D Raster Search			
Test Metric	Processing Time, Ratio of valid/invalid Spawns			
Observed Behavior	Processing time and spawn detection ratio depending step size			

Table 6.12: Test 6 Setup for performance test of 2D raster search in complex environment.

Table 6.13 presents the results of the performance test applying the 2D raster search to the complex environment. Again, the data shows, that the percentage of valid and invalid locations do not add up to 100%. As the design of this scene does not include terrain towards the outer edge of the environment, no spawn locations were assessed for positions that were far out in the sea. The processing time also behaves similarly to the previous test, where the time increases with smaller step size and does not increase with the same magnitude as the 3D search time. The ratio of invalid to valid spawn locations also behaves similarly to the ratio found with test 4. The only significant difference in the ratio is between test 4.3 and test 6.3. This is most certainly caused by the 3D algorithm missing a significant amount of viable spawn locations as the terrain is located between steps.

Test #	**Test 6.1**	**Test 6.2**	**Test 6.3**	**Test 6.4**
Step Size [m]	1 m	1.5 m	2 m	2.5 m
Nr. of Checks	≈ 1,640,000	≈ 729,000	≈ 411,000	≈ 263,000
Nr. of Valid Locations	≈ 618,000	≈ 276,000	≈ 155,000	≈ 99,200
Valid Locations [%]	37.7 %	37.8 %	37.6 %	37.7 %
Invalid Locations [%]	35.5 %	35.5 %	35.3 %	35.2 %
Total Processing Time [s]	**100.2 s**	**45.1 s**	**24.5 s**	**16.7 s**
Time per 1000 checks [ms]	**61.10 ms**	**61.79 ms**	**59.69 ms**	**63.61 ms**
Ratio Invalid/Valid	**0.943**	**0.938**	**0.938**	**0.935**

Table 6.13: Test 6 results of performance test of 2D raster search in complex environment.

6.2.2.2 Functionality Tests

As the performance of the 2D raster search seems to be preferable over the 3D search, I also wanted to analyze this algorithm's functionality. For this purpose, I used the same approach as for the previous functionality test and executed the algorithm in both environments and present the result with markers in the scene view. Figure 6.6 shows some special cases in the low-complexity environment and how the algorithm classifies the spawn locations. The examined locations are indicated with green markers for valid locations and red markers for invalid spawn locations.

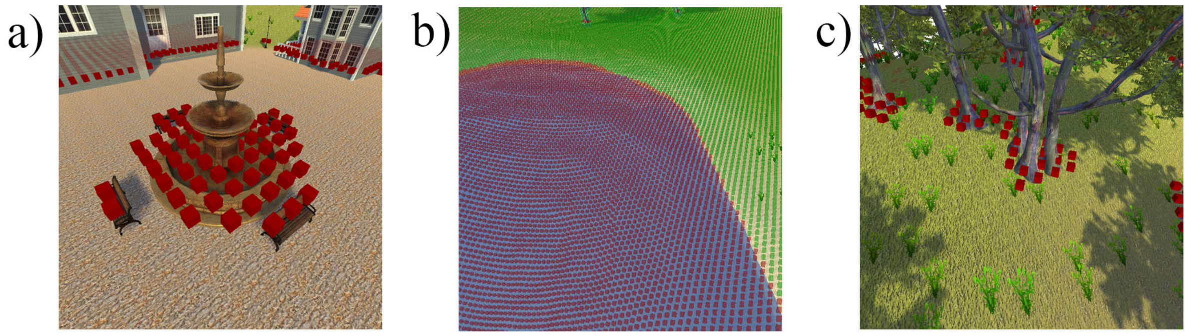


Figure 6.6: Overview of the classification of spawn locations by the 2D-raster-search algorithm in the low-complexity environment. a) Collision with fountain in town. b) Spawn classification for open terrain and lake. c) Collision detection in dense forest.

As shown by the special cases in Figure 6.6, the decisions made by the 2D search algorithm also correctly classified the spawn locations. As shown in example a), locations around the objects in the town center are correctly classified as invalid locations. In example b), tested position in the water are correctly marked as invalid while locations on terrain are marked as valid. Example c) shows the detection of small objects such as trees, which are also correctly identified.

The algorithm was also applied to the complex environment. Figure 6.7 provides an overview of the algorithm's classification of spawn locations in different situations.

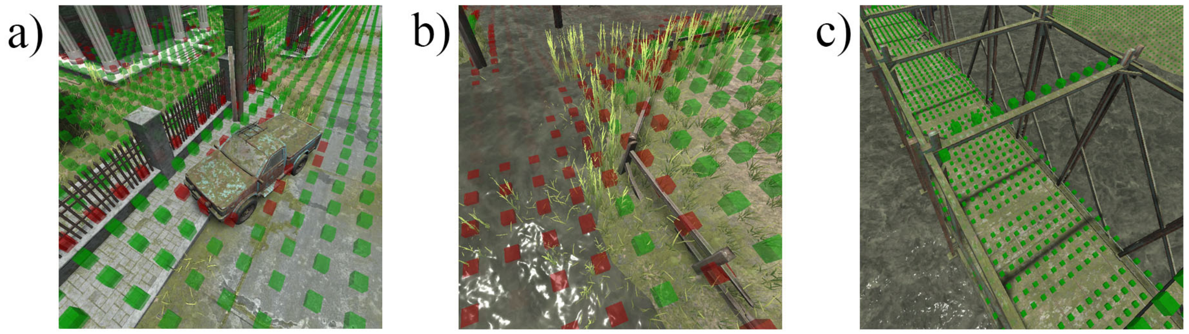


Figure 6.7: Overview of the classification of spawn locations by the 2D-raster-search algorithm in the complex environment. a) Collision detection in town. b) Water and collision detection with small objects. c) Spawn detection on bridge.

The algorithm correctly handles objects such as cars or fences and marks them as invalid locations as shown in example a). When comparing example b) to the results of the 3D search, it can be seen that no spawn locations above the fence are marked as valid. This happens because the algorithm only checks locations that are a defined distance above the terrain. Example c) shows a similar outcome as the result of the 3D search algorithm. However, after a location on the frame was checked, the location on the path of the bridge is not checked again. This is caused by the issue that a ray cast only returns the collider once. Therefore, the collider is stored when when the ray hits the frame and not again for the path on the bridge's surface. Similar to the previous functionality test with the 3D raster search, this issue could be solved by modeling the bridges differently to include separate colliders for the path and the frame.

6.2.3 Comparison of 2D- and 3D Raster Search Performance

As shown by the performance tests for both search algorithms, the processing time for the 3-dimensional search increases more drastically with lower step size than the 2-dimensional search. While the processing time per single check is longer for the 2D search, the quadratic increase in steps taken for the search results in a lower processing time than the cubic increase of the 3D search algorithm. Therefore, at a certain point, the 2-dimensional search will be more efficient than the 3-dimensional search algorithm.

To compare the effect of step sizes and the number of steps with the required processing time, tests 3 to 6 were repeated with smaller increments of step size. With the additional data, a set of graphs were created which show the increase of processing time depending on the number of steps taken. Figure 6.8 shows graphs comparing the processing time depending on step size and number of total steps.

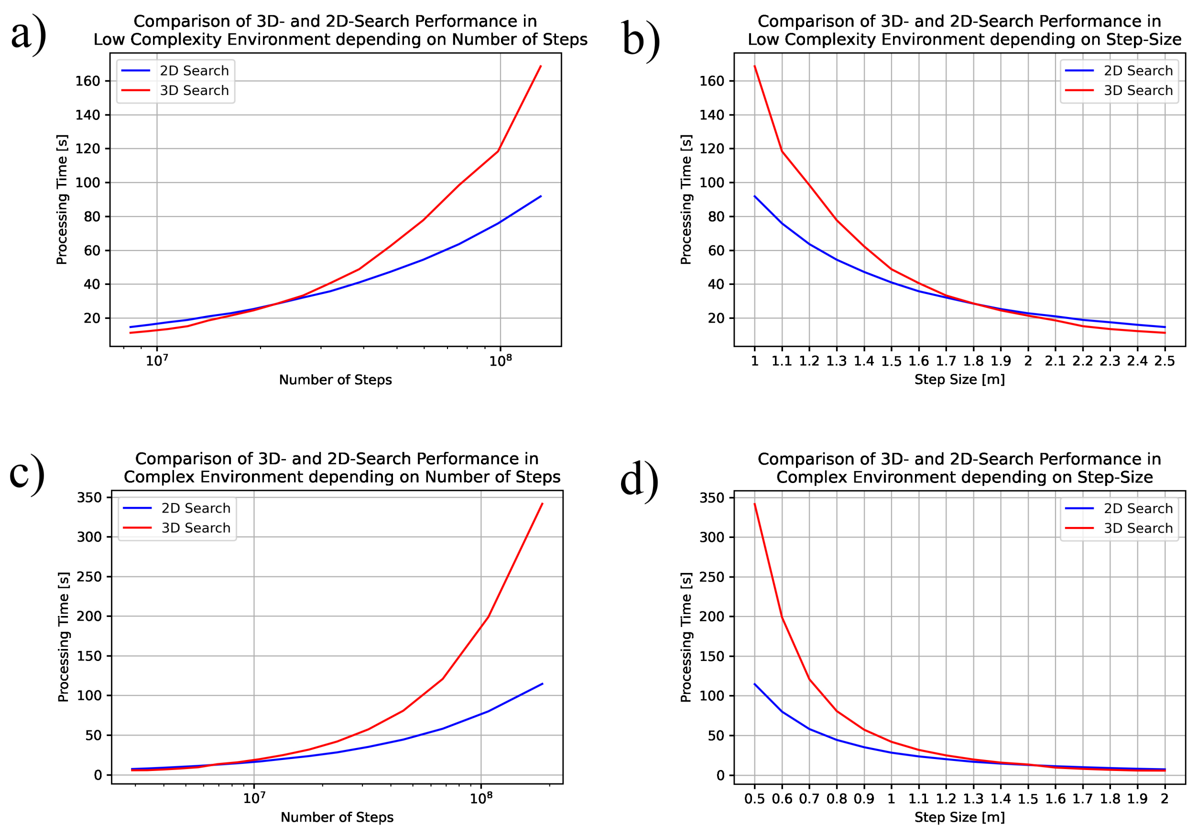


Figure 6.8: Performance graphs of 2D- and 3D-Search. a) Low-Complexity Environment: Processing Time vs. Number of Steps. b) Low-Complexity Environment: Processing Time vs. Step Size. c) Complex Environment: Processing Time vs. Number of Steps. d) Complex Environment: Processing Time vs. Step Size.

By comparing the graphs, information about the break-even point of the two algorithms can be gained. As shown in graphs a) and b), the processing time for both algorithm in the low-complexity environment is approximately equal at about 12.5 million steps which corresponds to a step size of about 1.8 meters. When comparing the algorithms' performances in the complex environment as shown in graphs c) and d), the break-even point is at approximately 20 million steps and a step size of about 1.8 meters. After this point, and increased number of positions to check favors the use of the 2D-raster-search algorithm.

The mentioned issue with the 3D search algorithm is the possibility to skip the terrain because of the defined step size. This issue can have a considerable impact when choosing large step sizes, especially if the step size is larger than the maximum height above ground. Because the 2D search algorithm does only consider the height of the terrain hit, such an issue does not occur. Figure 6.9 compares the result when choosing a step size of 2.5 meters and a height parameter of 1 meter for the 2D- and the 3D search algorithm. As shown in the picture, the 3D algorithm skips the terrain at certain positions, as the terrain is located between two steps, but not within the height. Therefore, some feasible locations are missed when performing a 3D raster search. Compared to the 3D algorithm, the 2D search algorithm finds all locations in the same area. As the maximum spawn height can not always be increased with the step size, as it could result in a large drop when starting the simulation, a solution of independent step-sizes for the XZ-plane and Y-axis was implemented.

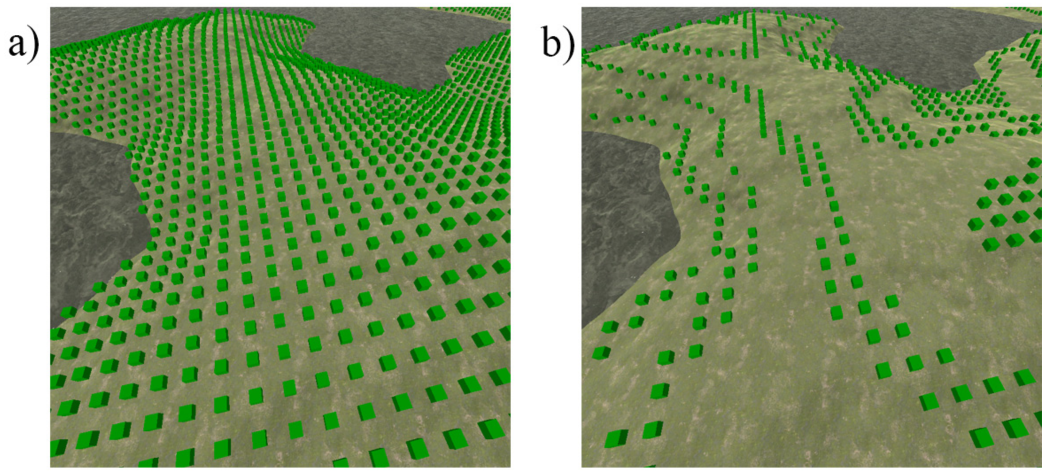


Figure 6.9: Comparison of a) 2D-search and b) 3D search algorithm and the resulting valid spawn locations. Step size: 2.5 meter, spawn height: 1 meter.

According to the results of the tests conducted, it can be assumed that the complexity, size and shape of the environment determine the exact number of steps at which the 2D raster search becomes more efficient. Therefore, the 3D raster search might still be a viable solution to search for valid spawn locations in a broader grid. The advantage of the 2D search is the ability to search a narrow grid and yield a large set of viable spawn locations within a reasonable calculation time. Considering the increased processing time of the 3D search and the issue of potentially missing feasible locations due to step size and maximum spawn height, it is concluded that the 2D raster search provides generally better results.

6.2.4 Random Search

In contrast to the raster-search-algorithms, which examine the environment in a structured manner, this random search algorithm assesses randomly chosen locations within the defined boundaries. This algorithm can not directly be compared to the other algorithms in terms of efficiency to find all viable locations. However, the random search may be a good option to randomly generate a defined number of spawn locations without first searching the entire environment. Therefore, the performance tests are structured differently.

6.2.4.1 Performance Test

The key performance to test for this algorithm is the time to find a set number of valid spawn locations. As the time to find a new spawn location is nondeterministic, it is not possible to run a single experiment and receive significant data. Therefore, the same test parameters are used to run multiple tests and visualize the performance for each run. In a further step, more tests are performed to gain an average time required to find the determined number of valid spawn locations.

Low-Complexity Environment

Again, the first test is conducted on the less complex environment. The test is repeated 5 times to present overlaying graphs of the time required to find new spawn locations. In a next step, the test is performed 20 times to gain an average total time to find the set number of spawn locations. Table 6.14 presents the setup for test 7.

Test #	**Test 7.1**	**Test 7.2**
Nr. of Tests	5	20
Test Area (x, y, z)	From (0, 31, 0) to (500, 124, 500)	
Min. Distance between Checks [m]	2.5 m	
Nr. of Spawns to Find	2,000	
Limit Consecutive Failures	100,000	
Max. Height above Ground [m]	1.0 m	
Test Environment	Low-Complexity Environment	
Test Algorithm	Random Search	
Test Metric	Completion time and time per per spawn found.	
Observed Behavior	Total time to find all spawns and compare the time between new spawns found.	

Table 6.14: Test 7 Setup for performance test of random search algorithm in low-complexity environment.

For test 7.1, the search was performed 5 times using the same settings. Figure 6.10 shows the time required to find all searched valid spawn locations. As shown by the graphs, each run behaves slightly differently. This is caused by the nondeterministic behavior of the search algorithm. However, the differences between the completion times are not as large as expected. Furthermore, each run yielded the desired number of 2,000 valid spawn locations.

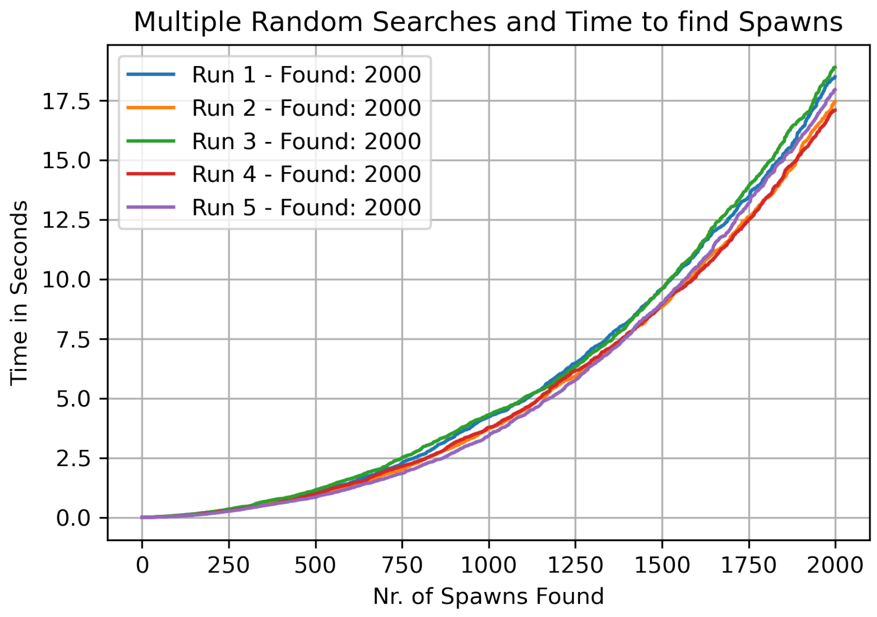


Figure 6.10: Test 7.1 results showing the time required to find 2,000 valid spawn locations in the low-complexity environment.

An interesting observations is that the time between the detection of valid locations seems to follow polynomial growth. The main impact on the processing time for this algorithm is the test for nearby checked locations. As for each checked location the list grows, the search for potentially nearby locations gets more time consuming with every iteration. In order to test this hypothesis, a separate test without the implementation of the distance check. Therefore, the binary search of a list of already assessed locations is skipped and the processing time per new location should not increase. Figure 6.11 shows the result of this test. As shown by the graphs, which represent the time to find a number of locations, the time to find a new locations does not seem to increase. The time required to find 2,000 feasible locations is also considerably lower than with the included distance check. Therefore it is assumed, that the distance check is one of the main factors for the processing time of this algorithm.

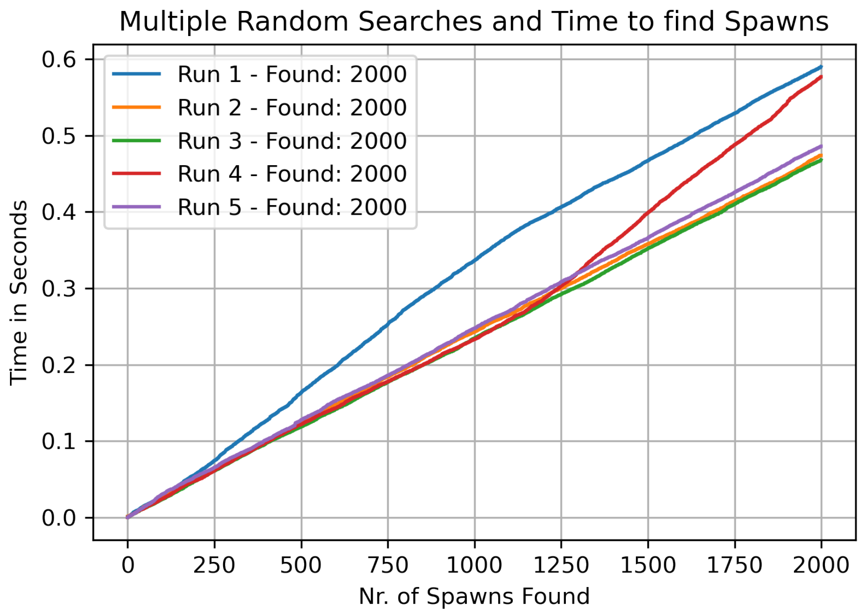


Figure 6.11: Test of the random search algorithm without distance check for previously checked locations.

To analyze the average completion time of the algorithm with the defined properties as shown in Table 6.14, a test with 20 consecutive runs was performed. Table 6.15 shows the average completion time as well as the variance of the data set. The results show, that the completion time to find a rather large number of spawn positions is low compared to a full search using the raster search algorithms. Furthermore, the variance of the data set lies within an acceptable range considering the quantity of found locations spread over the entire environment.

Test #	**Test 7.2**
Environment	Low-Complexity Environment
Nr. of Runs	20
Total Completion Time [s]	**376.5 s**
Average Time per Run [s]	**18.8 s**
Variance of Completion Time [s^2]	**1.03 s^2**

Table 6.15: Test 7.2 results of average completion time for random search algorithm in low-complexity environment.

To showcase the result of the random search algorithm, Figure 6.12 presents an overview of the spread of 2,000 valid spawn locations in the low-complexity environment. This overview shows that the discovered spawn locations are in fact random and do not seem to follow a certain trend.

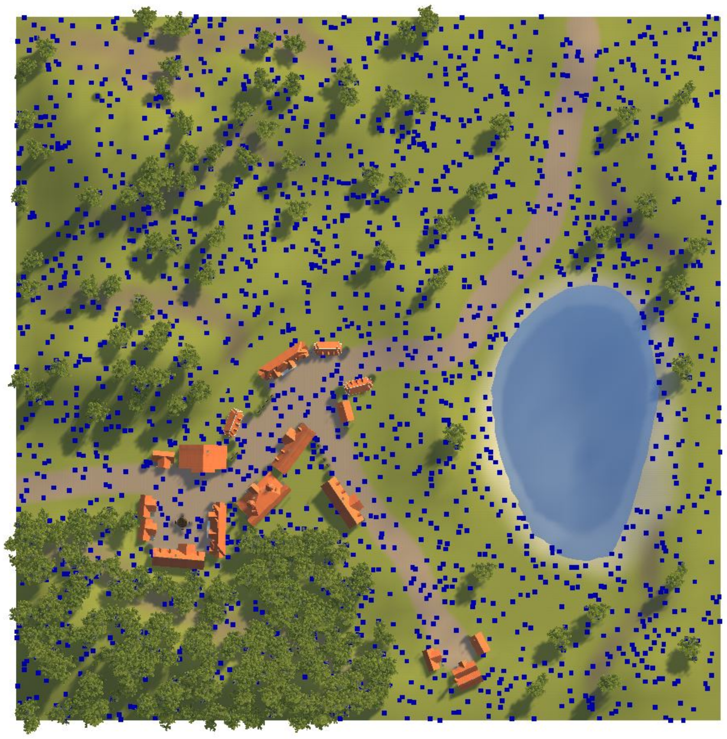


Figure 6.12: Overview of 2,000 random spawn location (blue) in the low-complexity environment.

Complex Environment

Analog to test 7, another experiment with the complex environment was conducted. This test was performed with the same parameters as for test 7 but the search area was adjusted to include the entire environment. Table 6.16 presents the setup parameters for the performance test using the random search algorithm within the complex environment.

Test #	**Test 8.1**	**Test 8.2**
Nr. of Tests	5	20
Test Area (x, y, z)	From (-128, 0, -212) to (1152, 80, 1067)	
Min. Distance between Checks [m]	2.5 m	
Nr. of Spawns to Find	2,000	
Limit Consecutive Failures	100,000	
Max. Height above Ground [m]	1.0 m	
Test Environment	Complex Environment	
Test Algorithm	Random Search	
Test Metric	Completion time and time per per spawn found.	
Observed Behavior	Total time to find all spawns and compare the time between new spawns found.	

Table 6.16: Test 8 Setup for performance test of random search algorithm in complex environment.

To visually present the differences between separate test runs, the results from test 8.1 are shown in Figure 6.13. The graphs clearly show a variance in completion time over the set of 5 test runs. Similar to test 7.1, all runs follow the same trend and found the required 2,000 spawn locations. Again, the total processing time varies from run to run.

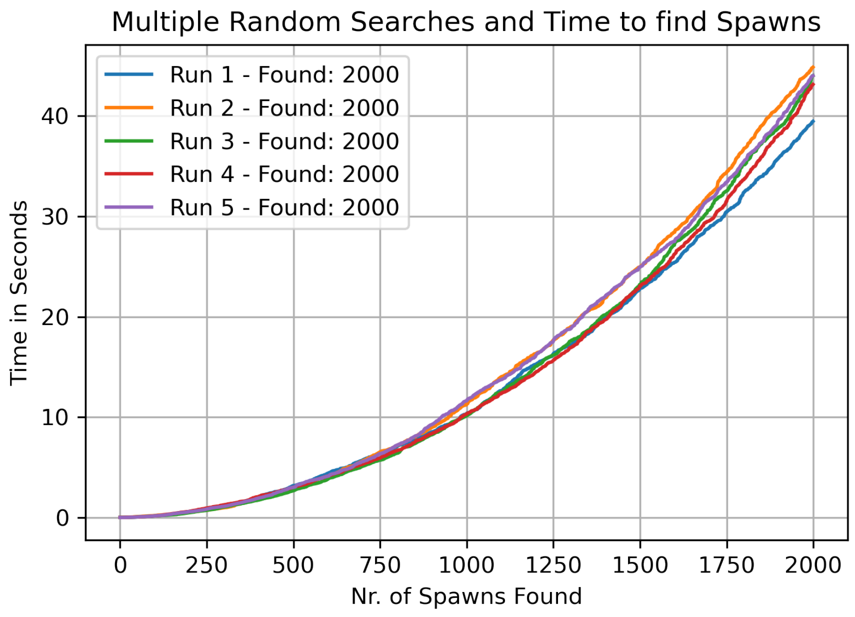


Figure 6.13: Test 8.1 results showing the time required to find 2,000 valid spawn locations in the complex environment.

Performing test runs with the same settings as in test 8.1, a test with 20 consecutive runs was performed. The resulting total completion time, average run time and the variance of the data-set are summarized in Table 6.17. With an increased size of the environment, the total completion time is larger than the time required to search the less complex environment. The main factor for the search time, using a random search algorithm, is the probability to find a valid location. Considering the results from test 3 (Table 6.7) and test 4 (Table 6.9), the probability to find a valid spawn location in the complex environment ($\approx 0.4\%$) is approximately half of the probability to find a viable location in the less complex environment ($\approx 0.95\%$). Therefore, it is reasonable that the runtime to find the equal amount of valid locations is also roughly doubled.

Test #	**Test 8.2**
Environment	Complex Environment
Nr. of Runs	20
Total Completion Time [s]	**852.5 s**
Average Time per Run [s]	**42.6 s**
Variance of Completion Time [s^2]	**3.05 s^2**

Table 6.17: Test 8.2 results of average completion time for random search algorithm in complex environment.

Chapter 7: Conclusion

7.1 Summary

Using Unity as a physics simulation platform has been well established in industry and research. The capability of the implemented physics engine and the accessibility of the development platform provide a convenient environment to develop simulations for autonomous robots. An approach to implement multiple robots into a simulation can help with the development of multi-agent simulations and performing tests with multi-agent control algorithms. In this thesis, I present three approaches to generate sets of viable spawn locations for such experiments.

The implemented algorithms provide an example for simple approaches to search for all feasible spawn locations within given search parameters, or to find a defined number of spawn locations which are spread randomly throughout the defined search perimeter. With the introduction of the 2-dimensional and 3-dimensional raster search algorithms, the user can have the entire environment explored and receive a text-based list of feasible spawn location coordinates. The resulting set of locations can then be processed to generate initial conditions for multiple simulations or simulations with multiple agents. By applying the random search algorithm, a predefined number of spawn locations are searched and returned. This algorithm does not return every possible location, but rather a set of random locations which are valid for spawning a robot in the simulation.

With this research, I applied the developed algorithms to various environments which vary in complexity and features present in the scene. The conducted performance analysis allowed to directly compare the 2D and 3D search algorithm in terms of processing time. With this comparison, it can be suggested to apply a 2-dimensional search for large environment or to generate a denser set of viable spawn locations. With the increased complexity per position check, the 3D search performs better for small sample sets. To generalize both algorithms performed well considering the size of the environment and the number of valid locations that are returned. The random search algorithm is preferable to directly generate a randomized set of initial position for a simulation. The performance of the random approach depends vastly on the design of the environment and the definition of the search perimeter, as the probability of finding a feasible locations within the perimeter factors into the search time of the algorithm.

The output of each method was assessed and the algorithms' classification of positions were examined for proper functionality of the code. Every algorithm managed to correctly classify spawn locations for their validity. Therefore, no location that was labeled as valid would result in a collision or otherwise unfeasible initial position for a robot simulation. With the parameter for maximum height above ground, for each algorithm, there are certain situations where a valid spawn point was located above a low-lying obstacle. Such conditions might result in unwanted behavior of the robot upon launching the physics simulation. Overall, the algorithms performed well and yielded satisfying results in both performance and functionality.

7.2 Contributions

With this thesis, I am proposed methods for generating feasible spawn positions, tested, and compared their performance applied to environments which vary in scale and complexity. The problem of searching feasible spawn locations has not been discussed to a great extent in available literature and there are only few approaches to find feasible locations. With the performed analysis of the implemented algorithms, a baseline for approaching the problem was created. The methods, used in this research, are easy to deploy in different environments and allow further research to be conducted with multi-agent simulations or to implement reinforcement learning approaches by performing a large number of simulations.

The results of the performance analysis suggest that the applied brute-force search algorithms already perform well within reasonable processing time. Each algorithm is executed with rather low time effort and can actively reduce the development time for generating multiple simulation scenarios. The performance analysis shows the advantage of the 2D search algorithm over the 3D search in any case where a large area with a dense search grid is explored, due to the reduced completion time. In order to generate a randomized list with a determined number of viable spawn locations, the random search algorithm can drastically reduces the search time by reducing the number of locations to check. However, the random search algorithm is not suitable to perform a full search of the environment.

7.3 Limitations

This research project only considers generating valid initial positions for robots. Therefore, there is no implementation of ROS or any robotics simulation itself. The result of the algorithms is a set of viable spawn-locations which could then be used to run autonomous robotic experiments. As it is assumed that the Unity simulation will start when the robot is placed and connected to ROS, it is desirable that the spawn locations are generated before running the Unity simulation. There is no real-time physics simulation applied and it is assumed that the environment is static. Collisions with dynamic objects would need to be double-checked once the simulation is launched. Therefore, if the environment for a robot simulation includes objects like moving cars and the robot should be spawned during runtime, a collision with the newly spawned robot could occur and the spawn location would need to be reevaluated. Additionally, spawn locations that were considered invalid when generating the set of spawn locations could be viable once a dynamic object moves. In the scope of this research, such issues are not resolved and would require further development.

Furthermore, this research only studies generating spawn positions for autonomous ground vehicles. Therefore, all spawn locations are expected to be either on the ground or close to the terrain which would result in only a small drop when launching the simulation. The terrain is also assumed to be in an acceptable shape that allows the robot to spawn on the ground. There is no check of the terrain's gradient to avoid a situation where the robot might roll over after spawning. The developed algorithms are intended to provide a good solution for most environments. However, there are always possible forms of terrain that could result in a location that is assumed suitable for the robot but results in an undesired physical issue once the real-time physics simulation is started. For this research, only the center position of the robot is used to check for the terrain height. Depending on the slope or geometry of the terrain, this approach can lead to issues in certain situations. An additional check for the terrain slop was not implemented but could potentially solve this issue.

7.4 Further Work

One critical issue that might occur when initiating the physics simulation, is the behavior of the robot when dropping to the ground. As the developed algorithms do not consider the shape and gradient of the terrain below the robot, unwanted issues such as a roll-over of the robot may occur. Further development could tackle this issue by analyzing the terrain in the area where the robot will make contact with the ground. Such an inspection could further increase the reliability of the generated spawn locations and reduce the number of failed simulation due to a demobilized robot. Combined with this addition, it might be considered to adjust the orientation of the robot to create the best-case scenario for each spawn location.

With the introduced search algorithms, only the spawn validity of each locations is assessed. While this guarantees a successful spawn procedure, it does not assure a successful simulation of the robot's mission. There are many situations where a spawn location can be classified as viable, but there would be no possible path to the goal locations for the robot's task. Additionally, there is no information about the quality of a spawn locations in terms of slope, maneuverability or the robot's vision at a given spawn location. Using the existing information of the designed environment, an interesting approach will be to determine the quality of a location and use this information to generate different simulation scenarios to assess the performance of the robot's planning algorithm.

In order to improve the search quality of the algorithms, a dynamic step size depending on the complexity of the environment in different regions could be added. This approach would require more information about the environment and could be used to increase the search density in regions with more obstacles and reduce the grids density in areas where less obstacles are located. Using such an approach can increase the quality of the search by finding locations even in a region which is densely occupied by obstacles, for example in a forest. By reducing the search density, less processing time will be required to search open areas where no or only few obstacles are expected.

Another aspect that was not considered in this research is the implementation in a dynamic environment. The introduced algorithms are not able to determine the validity of a location in an environment that may change over time. Such dynamic behavior presents a complex problem which would require a an improved algorithm which can handle dynamic objects in the environment.

Appendix A: Detailed Test Results

Test	**Test 1.1**	**Test 1.2**	**Test 1.3**	**Test 1.4**
Environment	Benchmark Environment	Benchmark Environment	Benchmark Environment	Benchmark Environment
Robot Collider Type	Complex	Complex	Complex	Complex
Obstacle Collider Type	None	None	None	None
Obstacle Arrangement	None	None	None	None
Nr. of Obstacles	0	0	0	0
Test Algorithm	3D Raster Search	3D Raster Search	3D Raster Search	3D Raster Search
Algorithm Settings:				
- Search Perimeter	**(0, 0, 0) to (9, 9, 499)**	**(0, 0, 0) to (9, 9, 999)**	**(0, 0, 0) to (9, 9, 4999)**	**(0, 0, 0) to (9, 9, 9999)**
- Step Size XZ	1 m	1 m	1 m	1 m
- Step Size Y	1 m	1 m	1 m	1 m
- Height Check	1.5 m	1.5 m	1.5 m	1.5 m
Results:				
- Nr. of Checked Positions	50,000	100,000	500,000	1,000,000
- Nr. of Valid Spawns	0	0	0	0
- Nr. of Invalid Spawns	0	0	0	0
- Nr. of Aerial Spawns	50,000	100,000	500,000	1,000,000
- Processing Time [ms]	**18.007 ms**	**34.030 ms**	**169.153 ms**	**341.974 ms**
- Processing Time per 1000 [ms]	**0.360 ms**	**0.340 ms**	**0.338 ms**	**0.342 ms**

Test	**Test 2.1**	**Test 2.2**	**Test 2.3**	**Test 2.4**
Environment	Benchmark Environment	Benchmark Environment	Benchmark Environment	Benchmark Environment
Robot Collider Type	**Box**	**Box**	**Complex**	**Complex**
Obstacle Collider Type	**Box**	**Complex**	**Box**	**Complex**
Obstacle Arrangement	(10, 10, 500)	(10, 10, 500)	(10, 10, 500)	(10, 10, 500)
Nr. of Obstacles	50,000	50,000	50,000	50,000
Test Algorithm	3D Raster Search	3D Raster Search	3D Raster Search	3D Raster Search
Algorithm Settings:				
- Search Perimeter	(0, 0, 0) to (9, 9, 499)	(0, 0, 0) to (9, 9, 499)	(0, 0, 0) to (9, 9, 499)	(0, 0, 0) to (9, 9, 499)
- Step Size XZ	1 m	1 m	1 m	1 m
- Step Size Y	1 m	1 m	1 m	1 m
- Height Check	1.5 m	1.5 m	1.5 m	1.5 m
Results:				
- Nr. of Checked Positions	50,000	50,000	50,000	50,000
- Nr. of Valid Spawns	0	0	0	0
- Nr. of Invalid Spawns	50,000	50,000	50,000	50,000
- Nr. of Aerial Spawns	0	0	0	0
- Processing Time - Run 1	10.457 s	10.766 s	9.484 s	10.657 s
- Processing Time - Run 2	10.557 s	11.009 s	9.556 s	11.005 s
- Processing Time - Run 3	10.596 s	10.839 s	9.430 s	11.439 s
- Processing Time - Run 4	10.488 s	10.801 s	9.723 s	11.190 s
- Processing Time - Run 5	10.728 s	10.775 s	10.224 s	11.260 s
- Processing Time - Run 6	10.696 s	10.901 s	10.722 s	11.224 s
- Processing Time - Run 7	10.560 s	10.960 s	10.779 s	11.245 s
- Processing Time - Run 8	10.660 s	11.009 s	10.622 s	11.221 s
- Processing Time - Run 9	10.654 s	10.987 s	10.812 s	11.231 s
- Processing Time - Run 10	10.916 s	10.892 s	10.493 s	11.050 s
- Average Processing Time [s]	**10.631 s**	**10.894 s**	**10.184 s**	**11.152 s**
- Avg. Processing Time per 1000 [ms]	**212.627 ms**	**217.877 ms**	**203.688 ms**	**223.042 ms**

Test	**Test 3.1**	**Test 3.2**	**Test 3.3**	**Test 3.4**
Environment	Low Complexity Environmnet	Low Complexity Environmnet	Low Complexity Environmnet	Low Complexity Environmnet
Robot Collider Type	Complex	Complex	Complex	Complex
Test Algorithm	3D Raster Search	3D Raster Search	3D Raster Search	3D Raster Search
Algorithm Settings:				
- Search Perimeter	(0, 31, 0) to (500, 124, 500)	(0, 31, 0) to (500, 124, 500)	(0, 31, 0) to (500, 124, 500)	(0, 31, 0) to (500, 124, 500)
- Step Size XZ	**0.5 m**	**1 m**	**1.5 m**	**2 m**
- Step Size Y	**0.5 m**	**1 m**	**1.5 m**	**2 m**
- Height Check	1 m	1 m	1 m	1 m
Results:				
- Nr. of Checked Positions	187,374,200	23,594,090	7,028,028	2,961,047
- Nr. of Valid Spawns	1,735,798	217,802	68,606	30,795
- Nr. of Invalid Spawns	260,086	32,467	10,392	4,817
- Nr. of Aerial Spawns	185,378,303	23,343,825	6,949,030	2,925,435
- Processing Time [s]	**363.372 s**	**47.365 s**	**14.000 s**	**6.606 s**
- Processing Time per 1000 [ms]	**1.939 ms**	**2.007 ms**	**1.992 ms**	**2.231 ms**

Test	**Test 4.1**	**Test 4.2**	**Test 4.3**	**Test 4.4**
Environment	Complex Environment	Complex Environment	Complex Environment	Complex Environment
Robot Collider Type	Complex	Complex	Complex	Complex
Test Algorithm	3D Raster Search	3D Raster Search	3D Raster Search	3D Raster Search
Algorithm Settings:				
- Search Perimeter	(-128, 0, -212) to (1152, 80, 1067)	(-128, 0, -212) to (1152, 80, 1067)	(-128, 0, -212) to (1152, 80, 1067)	(-128, 0, -212) to (1152, 80, 1067)
- Step Size XZ	**1 m**	**1.5 m**	**2 m**	**2.5 m**
- Step Size Y	**1 m**	**1.5 m**	**2 m**	**2.5 m**
- Height Check	1 m	1 m	1 m	1 m
Results:				
- Nr. of Checked Positions	132,814,100	39,383,060	16,846,120	8,684,577
- Nr. of Valid Spawns	538,963	125,019	73,634	40,121
- Nr. of Invalid Spawns	575,878	166,763	66,804	40,141
- Nr. of Aerial Spawns	131,699,239	39,045,166	16,679,402	8,587,386
- Processing Time [s]	**161.528 s**	**41.780 s**	**20.636 s**	**10.918 s**
- Processing Time per 1000 [ms]	**1.216 ms**	**1.061 ms**	**1.225 ms**	**1.257 ms**

Test	**Test 5.1**	**Test 5.2**	**Test 5.3**	**Test 5.4**
Environment	Low Complexity Environmnet	Low Complexity Environmnet	Low Complexity Environmnet	Low Complexity Environmnet
Robot Collider Type	Complex	Complex	Complex	Complex
Test Algorithm	2D Raster Search	2D Raster Search	2D Raster Search	2D Raster Search
Algorithm Settings:				
- Search Perimeter	(0, 0) to (500, 500)	(-128, 0, -212) to (1152, 80, 1067)	(-128, 0, -212) to (1152, 80, 1067)	(-128, 0, -212) to (1152, 80, 1067)
- Ray Perimeter	31 to 124	31 to 124	31 to 124	31 to 124
- Step Size XZ	**0.5 m**	**1 m**	**1.5 m**	**2 m**
- Y-Offset	1 m	1 m	1 m	1 m
Results:				
- Nr. of Checked Positions	1,002,001	251,001	111,556	63,001
- Nr. of Valid Spawns	895,994	223,985	99,990	55,980
- Nr. of Invalid Spawns	104,006	26,015	11,566	6,520
- Processing Time [s]	**114.672 s**	**28.596 s**	**12.647 s**	**7.214 s**
- Processing Time per 1000 [ms]	**114.443 ms**	**113.928 ms**	**113.369 ms**	**114.506 ms**

Test	**Test 6.1**	**Test 6.2**	**Test 6.3**	**Test 6.4**
Environment	Complex Environment	Complex Environment	Complex Environment	Complex Environment
Robot Collider Type	Complex	Complex	Complex	Complex
Test Algorithm	2D Raster Search	2D Raster Search	2D Raster Search	2D Raster Search
Algorithm Settings:				
- Search Perimeter	(-128, -212) to (1152, 1067)	(-128, -212) to (1152, 1067)	(-128, -212) to (1152, 1067)	(-128, -212) to (1152, 1067)
- Ray Perimeter	0 to 80	0 to 80	0 to 80	0 to 80
- Step Size XZ	**1 m**	**1.5 m**	**2 m**	**2.5 m**
- Y-Offset	1 m	1 m	1 m	1 m
Results:				
- Nr. of Checked Positions	1,639,680	729,316	410,881	263,169
- Nr. of Valid Spawns	618,089	275,742	154,688	99,228
- Nr. of Invalid Spawns	582,852	258,768	145,070	92,759
- Processing Time [s]	**100.190 s**	**45.064 s**	**24.526 s**	**16.740 s**
- Processing Time per 1000 [ms]	**61.103 ms**	**61.789 ms**	**59.691 ms**	**63.609 ms**

Test	Test 7.1	Test 7.2
Environment	Low Complexity Environment	Low Complexity Environment
Robot Collider Type	Complex	Complex
Test Algorithm	Random Search	Random Search
Algorithm Settings:		
- Nr. of Test Runs	5	20
- Search Perimeter	(0, 31, 0) to (500, 124, 500)	(0, 31, 0) to (500, 124, 500)
- Number of Spawns to find	2,000	2,000
- Min. Distance between Checks	**2.5 m**	**2.5 m**
- Height Check	1 m	1 m
- Limit of Consecutive Failures	100,000	100,000
Results:		
- Completion Time (Run 1) [s]	18.486 s	18.700 s
- Completion Time (Run 2) [s]	17.448 s	18.634 s
- Completion Time (Run 3) [s]	18.892 s	17.175 s
- Completion Time (Run 4) [s]	17.092 s	19.705 s
- Completion Time (Run 5) [s]	17.953 s	18.033 s
- Completion Time (Run 6) [s]		17.769 s
- Completion Time (Run 7) [s]		16.586 s
- Completion Time (Run 8) [s]		19.145 s
- Completion Time (Run 9) [s]		19.681 s
- Completion Time (Run 10) [s]		17.632 s
- Completion Time (Run 11) [s]		20.611 s
- Completion Time (Run 12) [s]		19.636 s
- Completion Time (Run 13) [s]		20.061 s
- Completion Time (Run 14) [s]		19.330 s
- Completion Time (Run 15) [s]		19.457 s
- Completion Time (Run 16) [s]		19.785 s
- Completion Time (Run 17) [s]		18.927 s
- Completion Time (Run 18) [s]		18.873 s
- Completion Time (Run 19) [s]		18.890 s
- Completion Time (Run 20) [s]		17.832 s
- Total Completion Time [s]	89.871 s	376.462 s
- Average Completion TIme [s]	**17.974 s**	**18.823 s**
- Variance of Completion Time [s^2]	**0.432 s^2**	**1.026 s^2**

Test	Test 8.1	Test 8.2
Environment	Complex Environment	Complex Environment
Robot Collider Type	Complex	Complex
Test Algorithm	Random Search	Random Search
Algorithm Settings:		
- Nr. of Test Runs	5	20
- Search Perimeter	(-128, 0, -212) to (1152, 80, 1067)	(-128, 0, -212) to (1152, 80, 1067)
- Number of Spawns to find	2,000	2,000
- Min. Distance between Checks	**2.5 m**	**2.5 m**
- Height Check	1 m	1 m
- Limit of Consecutive Failures	100,000	100,000
Results:		
- Completion Time (Run 1) [s]	39.474 s	42.095 s
- Completion Time (Run 2) [s]	44.863 s	45.335 s
- Completion Time (Run 3) [s]	44.038 s	42.997 s
- Completion Time (Run 4) [s]	43.15 s	43.613 s
- Completion Time (Run 5) [s]	44.027 s	41.970 s
- Completion Time (Run 6) [s]		40.759 s
- Completion Time (Run 7) [s]		42.938 s
- Completion Time (Run 8) [s]		44.739 s
- Completion Time (Run 9) [s]		41.364 s
- Completion Time (Run 10) [s]		42.255 s
- Completion Time (Run 11) [s]		43.144 s
- Completion Time (Run 12) [s]		44.286 s
- Completion Time (Run 13) [s]		40.505 s
- Completion Time (Run 14) [s]		37.906 s
- Completion Time (Run 15) [s]		41.288 s
- Completion Time (Run 16) [s]		43.336 s
- Completion Time (Run 17) [s]		44.753 s
- Completion Time (Run 18) [s]		44.545 s
- Completion Time (Run 19) [s]		43.193 s
- Completion Time (Run 20) [s]		41.439 s
- Total Completion Time [s]	215.552 s	852.460 s
- Average Completion TIme [s]	**43.110 s**	**42.623 s**
- Variance of Completion Time [s^2]	**3.599 s^2**	**3.049 s^2**

Appendix B: Code and Project Installation Guide

This guide will provide information for setting up the algorithms or my Unity project on your computer. I am providing a GitHub repository which contains the algorithms as well as a functioning Unity demo project which only includes the Training Environment. Due to license issues, I am not allowed to publicly share the entire Unity project, including the low complexity- and the complex environment. If interested, please reach out to me and I will be able to provide access to the full project folder used in this research project.

Contact: rafael.ropelato.msc@gmail.com

B.1 Demo Project Setup

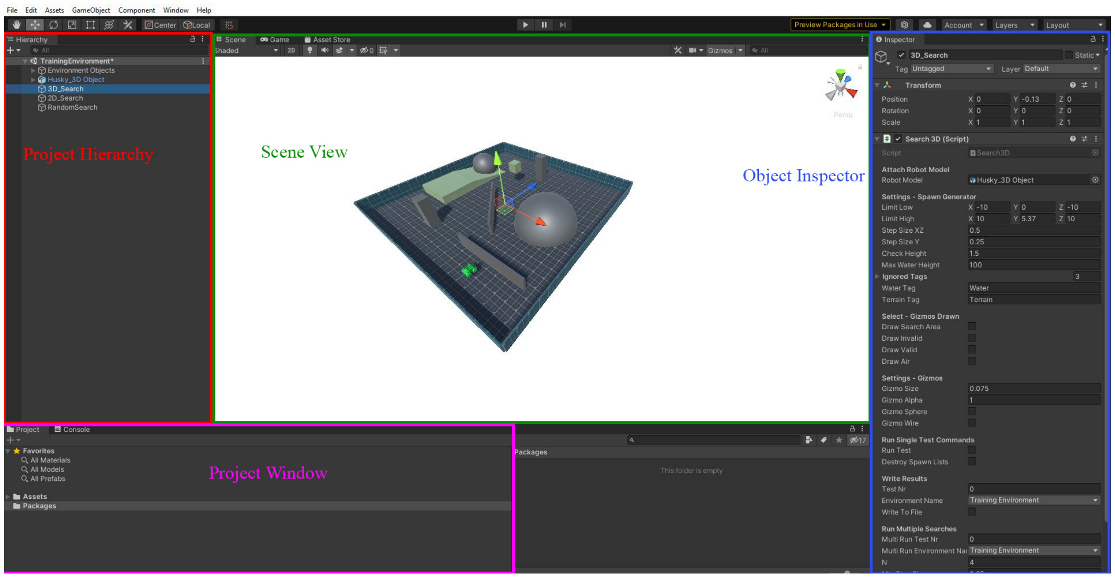
First, please clone the "Demo Project" folder from the following GitHub repository to your computer. The GitHub repository can be found under:

https://github.com/Rafael-Ropi/Ropelato-MSc-Thesis-Demo-Project

Launch Project

- Download and Install Unity Hub (https://unity3d.com/get-unity/download)
- Install Unity version **2020.3.24f1** (Other Unity versions might be compatible. Not tested!)
- Open Unity project folder in Unity Hub

Figure B.1 shows the windows within the Unity program as an overview for the following steps.



```
Inspector
3D_Search                                Static
Tag Untagged              Layer Default
Transform
Position                  X 0      Y -0.13   Z 0
Rotation                  X 0      Y 0       Z 0
Scale                     X 1      Y 1       Z 1
Search 3D (Script)
Script                    Search3D
Attach Robot Model
Robot Model               Husky_3D Object
Settings - Spawn Generator
Limit Low                 X -10    Y 0       Z -10
Limit High                X 10     Y 5.37    Z 10
Step Size XZ              0.5
Step Size Y               0.25
Check Height              1.5
Max Water Height          100
Ignored Tags                                 3
Water Tag                 Water
Terrain Tag               Terrain
Select - Gizmos Drawn
Draw Search Area          ☐
Draw Invalid              ☐
Draw Valid                ☐
Draw Air                  ☐
Settings - Gizmos
Gizmo Size                0.075
Gizmo Alpha               1
Gizmo Sphere              ☐
Gizmo Wire                ☐
Run Single Test Commands
Run Test                  ☐
Destroy Spawn Lists       ☐
Write Results
Test Nr                   0
Environment Name          Training Environment
Write To File             ☐
Run Multiple Searches
Multi Run Test Nr         0
Multi Run Environment Nai Training Environment
N                         4
```

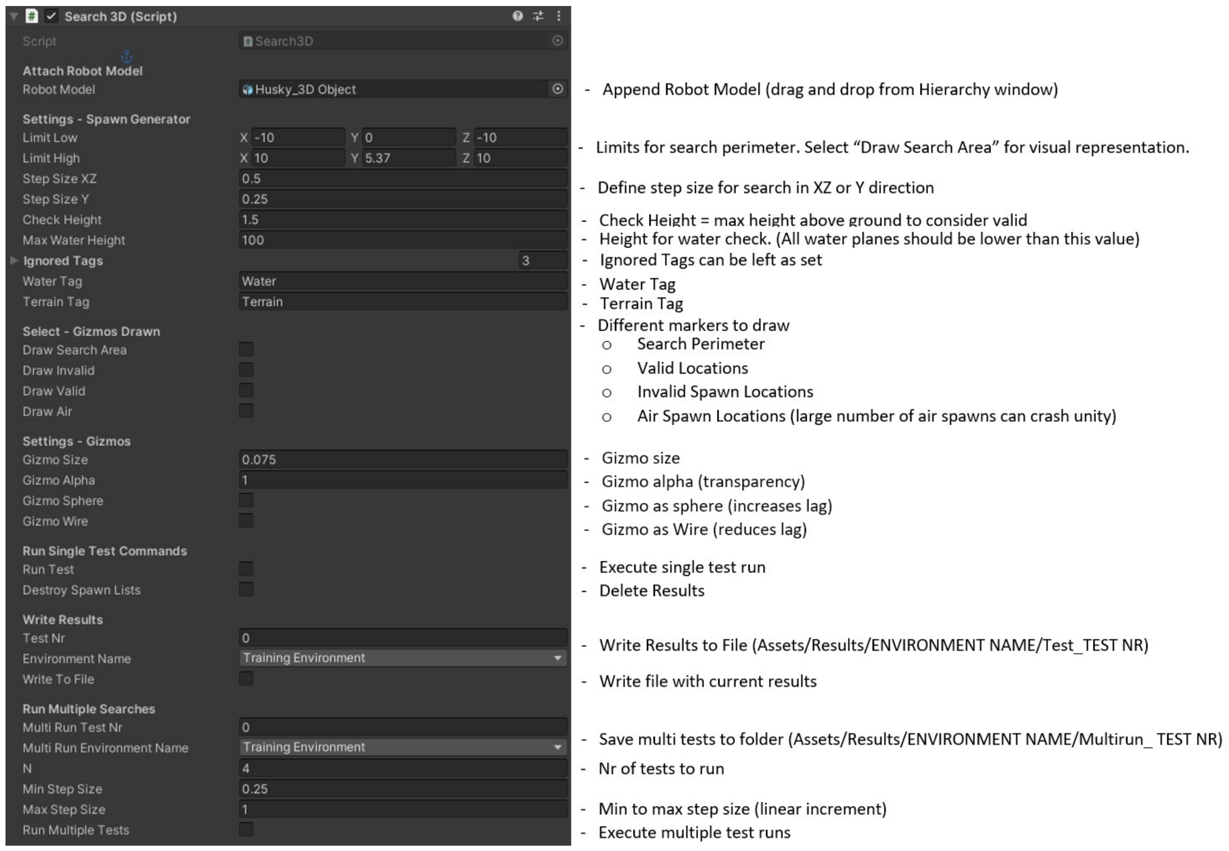
Figure B.1: Overview of the Unity window.

Launch Search Algorithms

A reference for the parameters to set is shown in Figure B.2.

- Once the Project was loaded, it should open the training environment. If not, select the scene in the Project Window under Assets/Scenes/Training Environment.
- The Project Hierarchy shows all GameObjects. This includes th 2D_Search, 3D_Search and RandomSearch object.
- Select any of the search algorithms.
- The parameters for the search algorithm can be set in the Object Inspector on the right-hand side.

- Set up the parameters and launch the algorithm by setting the RUN TEST boolean (check box) to $True$.



Search 3D (Script)

Script	Search3D	
Attach Robot Model		
Robot Model	Husky_3D Object	- Append Robot Model (drag and drop from Hierarchy window)
Settings - Spawn Generator		
Limit Low	X -10 Y 0 Z -10	- Limits for search perimeter. Select "Draw Search Area" for visual representation.
Limit High	X 10 Y 5.37 Z 10	
Step Size XZ	0.5	- Define step size for search in XZ or Y direction
Step Size Y	0.25	
Check Height	1.5	- Check Height = max height above ground to consider valid
Max Water Height	100	- Height for water check. (All water planes should be lower than this value)
Ignored Tags	3	- Ignored Tags can be left as set
Water Tag	Water	- Water Tag
Terrain Tag	Terrain	- Terrain Tag
Select - Gizmos Drawn		- Different markers to draw
Draw Search Area		○ Search Perimeter
Draw Invalid		○ Valid Locations
Draw Valid		○ Invalid Spawn Locations
Draw Air		○ Air Spawn Locations (large number of air spawns can crash unity)
Settings - Gizmos		
Gizmo Size	0.075	- Gizmo size
Gizmo Alpha	1	- Gizmo alpha (transparency)
Gizmo Sphere		- Gizmo as sphere (increases lag)
Gizmo Wire		- Gizmo as Wire (reduces lag)
Run Single Test Commands		
Run Test		- Execute single test run
Destroy Spawn Lists		- Delete Results
Write Results		
Test Nr	0	- Write Results to File (Assets/Results/ENVIRONMENT NAME/Test_TEST NR)
Environment Name	Training Environment	
Write To File		- Write file with current results
Run Multiple Searches		
Multi Run Test Nr	0	- Save multi tests to folder (Assets/Results/ENVIRONMENT NAME/Multirun_ TEST NR)
Multi Run Environment Name	Training Environment	
N	4	- Nr of tests to run
Min Step Size	0.25	- Min to max step size (linear increment)
Max Step Size	1	
Run Multiple Tests		- Execute multiple test runs

Figure B.2: Description of the search parameters for the 3D search algorithm.

<u>Custom Environment</u>

In order to create your own environment, some guidelines have to be followed.

- All objects that can cause a collision need to have a collider attached
- All objects which should server as spawnable terrain need to have the tag "Terrain". Can be set in the top-left of the Object Inspector.
- All water planes need to have the tag "Water".

Custom Robot

In order to use a different robot or object, set it up as follows.

- Import object.
- Add script in object inspector. ”Add Component” $\rightarrow$ search ”Feasibility Check (Script)”.
- Append a ”Rigid Body” and a ”Collider” object in the object inspector.

  If mesh collider, make it ”Convex”.

  Make collider ”Is Trigger”.

  Make rigid body ”Is Kinematic”.
- Attach robot object to the search algorithms. (Drag-and-drop robot object to Robot Model variable)
- (OPTIONAL) Attach ”Colored Collision Detector (Script)” to robot object. This changes the color of the robot to green or red depending on collision status in scene view.

Appendix C: List of Used Assets

The following lists present all assets that were used to create the environments for this research project. All assets are free-to-use assets from the Unity Asset Store [10].

Low Complexity Environment

Asset Name	**Publisher**	**Application**
Fountain Prop	Thunderent	Fountain for Town
Realistic Tree 9 [Rainbow Tree]	Pixel Games	Trees for Forest
GAZ Street Props	Helsssoo	Benches and Street Lamps
House Pack	Mehdi Rabiee	Buildings in Town
Terrain Sample Asset Pack	Unity Technologies	Terrain Creation Tools
Grass And Flowers Pack 1	Vladislav Pochezhertsev	Flowers
Outdoor Ground Textures	A dog's life software	Terrain Textures

Complex Environment

Asset Name	**Publisher**	**Application**
Flooded Grounds	Sandro T	Complex Environment
Conifers [BOTD]	forst	Detailed Trees

Bibliography

[1] “ISO 8373:2012(en), Robots and robotic devices — Vocabulary.” https://www.iso.org/obp/ui/#iso:std:iso:8373:ed-2:v1:en (accessed Feb. 17, 2022).

[2] Unity Technologies, “Unity Real-Time Development Platform — 3D, 2D VR & AR Engine.” https://unity.com/ (accessed Feb. 21, 2022).

[3] “PhysX SDK,” NVIDIA Developer, Nov. 28, 2018. https://developer.nvidia.com/physx-sdk (accessed Feb. 21, 2022).

[4] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R. & Ng, A. ROS: an open-source Robot Operating System. *ICRA Workshop On Open Source Software*. **3** (2009,1)

[5] C. Bartneck, M. Soucy, K. Fleuret, and E. B. Sandoval, “The robot engine 2014; Making the unity 3D game engine work for HRI,” in 2015 24th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN), Kobe, Japan, Aug. 2015, pp. 431–437.

[6] Juliani, A., Berges, V., Vckay, E., Gao, Y., Henry, H., Mattar, M. & Lange, D. Unity: A General Platform for Intelligent Agents. (2018,9)

[7] A. Hussein, A. Diaz-Alvarez, J. M. Armingol, and C. Olaverri-Monreal, “3DCoAutoSim: Simulator for Cooperative ADAS and Automated Vehicles,” in 2018 21st International Conference on Intelligent Transportation Systems (ITSC), Maui, HI, Nov. 2018, pp. 3014–3019. doi: 10.1109/ITSC.2018.8569512.

[8] Unity Technologies, “Unity - Manual: Creating and Using Scripts.” https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html (accessed Mar. 30, 2022).

[9] Blender Foundation, “blender.org - Home of the Blender project - Free and Open 3D Creation Software,” blender.org. https://www.blender.org/ (accessed Mar. 30, 2022).

[10] “Unity Asset Store - The Best Assets for Game Making.” https://assetstore.unity.com/ (accessed Mar. 30, 2022).

[11] Unity Technologies, “Unity - Manual: Colliders.” https://docs.unity3d.com/Manual/CollidersOverview.html (accessed Mar. 30, 2022).

[12] Unity Technologies, “Unity - Scripting API: Physics.Raycast.” https://docs.unity3d.com/ScriptReference/Physics.Raycast.html (accessed Mar. 30, 2022).

[13] Unity Technologies, “Unity - Scripting API: Physics.OverlapBox.” https://docs.unity3d.com/ScriptReference/Physics.OverlapBox.html (accessed Mar. 30, 2022).

[14] “Physics in scene editor mode? - Unity Answers.” https://answers.unity.com/questions/158766/physics-in-scene-editor-mode.html?childToView=1672642#answer-1672642 (accessed Mar. 30, 2022).

[15] R. H. F. Jackson, P. T. Boggs, S. G. Nash, and S. Powell, “Guidelines for reporting results of computational experiments. Report of the ad hoc committee,” Mathematical Programming, vol. 49, no. 1–3, pp. 413–425, Nov. 1990, doi: 10.1007/BF01588801.

[16] N. G. Hall and M. E. Posner, “Generating experimental data for computational testing with machine scheduling applications,” Operations Research, vol. 49, no. 6, pp. 854–865, Dec. 2001.

[17] Unity Technologies, “Unity - Manual: Mesh Collider.” https://docs.unity3d.com/Manual/class-MeshCollider.html (accessed Apr. 04, 2022).

[18] “Husky UGV - Outdoor Field Research Robot by Clearpath,” Clearpath Robotics. https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/ (accessed Mar. 30, 2022).