FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Una Adilović

Playing a 3D Tunnel Game Using Reinforcement Learning

Department of Software and Computer Science Education (KSVI)

Supervisor of the bachelor thesis:	Adam Dingle, M.Sc.
Study programme:	Computer Science
Study branch:	Artificial Intelligence

Prague 2023

I would like to express my heartfelt appreciation to my supervisor, Adam Dingle, for his invaluable guidance and support throughout my academic journey, beginning with our first Programming 1 class and continuing through to the completion of this project. His patience and assistance were greatly appreciated and played a significant role in the success of this work.

In addition, I am deeply thankful to my mother for her unwavering belief in me and to my grandparents who made it possible for me to be here. Their contributions are greatly appreciated and will never be forgotten.

Title: Playing a 3D Tunnel Game Using Reinforcement Learning

Author: Una Adilović

Department: Department of Software and Computer Science Education (KSVI)

Supervisor: Adam Dingle, M.Sc., Department of Software and Computer Science Education (KSVI)

Abstract: Tunnel games are a type of 3D video game in which the player moves through a tunnel and tries to avoid obstacles by rotating around the axis of the tunnel. These games often involve fast-paced gameplay and require quick reflexes and spatial awareness to navigate through the tunnel successfully. The aim of this thesis is to explore the representation of a tunnel game in a discrete manner and to compare various reinforcement learning algorithms in this context. The objective is to evaluate the performance of these algorithms in a game setting and identify potential strengths and limitations. The results of this study may offer insights on the application of discrete tabular methods in the development of AI agents for other continuous games. (ADD RESULT)

Keywords: tunnel game, reinforcement learning, artificial intelligence, algorithms

Contents

Introduction

Reinforcement learning is a subfield of machine learning that aims to train agents to make decisions that will maximize a reward signal [Sutton and Barto, 2018]. This approach has been widely applied in the field of artificial intelligence, particularly in the context of training agents to play games. In a game setting, an agent's actions can be evaluated based on their impact on the agent's score or likelihood of winning. Through the process of reinforcement learning, the agent learns to make strategic decisions that maximize its reward by receiving positive reinforcement for good moves and negative reinforcement for suboptimal moves. This allows the agent to adapt and improve its performance over time as it plays the game. Research on reinforcement learning in games has demonstrated its effectiveness in a variety of contexts, including board games, video games, and real-time strategy games.

In the field of artificial intelligence, games can be classified as either continuous or discrete based on the nature of the action space and state space. Continuous games have a continuous action space, meaning that the possible actions an agent can take are not limited to a fixed set of options, but can vary continuously within a certain range. In contrast, discrete games have a discrete action space, meaning that the possible actions are limited to a fixed set of options. Continuous games are often characterized by a high-dimensional state space, as they may involve a large number of variables that describe the game state. Discrete games, on the other hand, typically have a lower-dimensional state space, as the number of possible states is limited by the discrete action space. In general, continuous games are more challenging to model and solve than discrete games, as they require more complex decision-making algorithms and may require more computational resources.

In this thesis, we will investigate the application of reinforcement learning to train agents to play a continuous 3D tunnel game. The continuous game environment will be discretized into a set of states, and different reinforcement learning algorithms will be applied to train agents to play the game. The goal of this study is to determine whether it is possible for any of the agents to win the whole game, and to compare the performance of different agents that use different reinforcement learning algorithms.

The results of this study will contribute to the understanding of the potential of reinforcement learning for training agents to use discreate algorithms in a naturally continuous environment, and to provide insight into the strengths and weaknesses of different reinforcement learning algorithms in this context.

1. Game Design

1.1 Player and Movement

The game that will be analysed is called “Space-run” and it involves attempting to accumulate the highest score possible by navigating through three distinct tunnels while avoiding various obstacles. The game is endless in nature, as the speed increases each time the player successfully completes all three tunnels. It is worth noting that, in designing this game, I was inspired by the pre-existing “Tunnel rush”.

1.2 Player and Movement

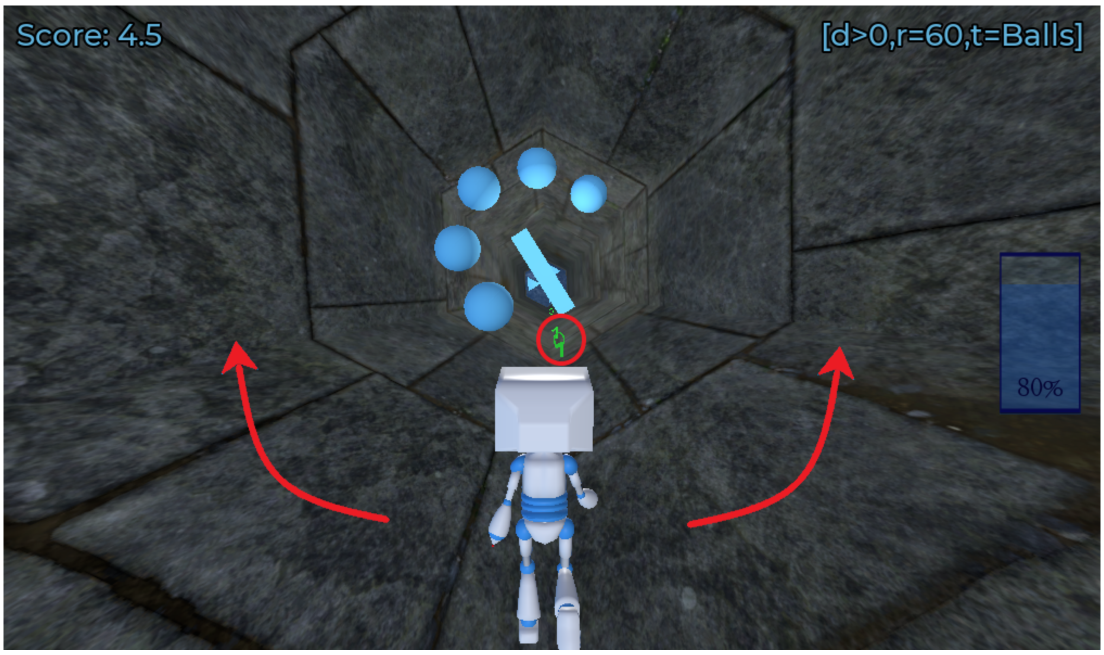

Figure 1.1: Movement

In “Space-run” the player assumes control of a character named Hans (see Figure 1.2) which continually advances at a constant speed through the tunnels. To navigate through the game, the player must use the left and right arrow keys to rotate the current tunnel and avoid obstacles (as shown in Figure 1.1). In addition to these lateral movements, Hans also has the ability to shoot bullets (also shown in Figure 1.1) by pressing the space key, which can be used to defeat certain in-game creatures and earn a higher score. The player must utilize these abilities in order to progress through the game and achieve a high score.

Figure 1.2: Hans

1.3 Obstacles

As previously stated, the player must navigate through various obstacles in the game. These obstacles can be divided into three distinct categories, and each tunnel contains a unique subset of them. In the subsequent sections, we will delve deeper into these categories in order to better understand the challenges faced by the player.

1.3.1 Traps

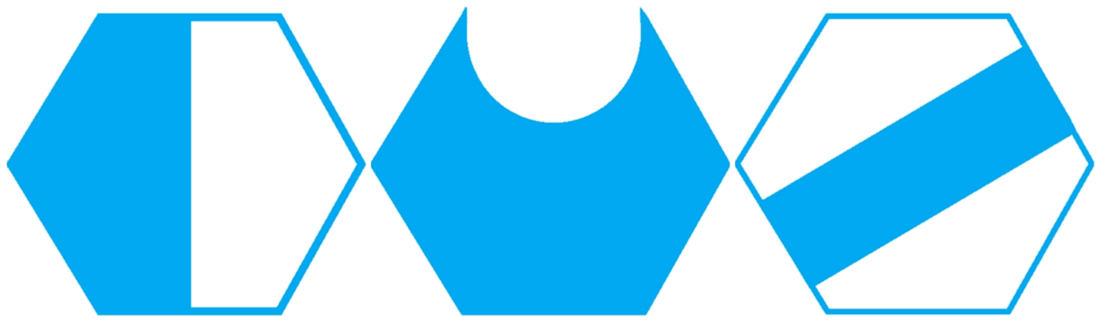

Figure 1.3: Trap examples

As depicted in Figure 1.3, a selection of the various trap types that the character Hans must avoid is presented. These traps, of which there are a total of 10, vary in their level of difficulty and can be either static or animated. These traps can be encountered in any of the three tunnels, and if the player fails to successfully evade them, they result in an instant death.

1.3.2 Bugs

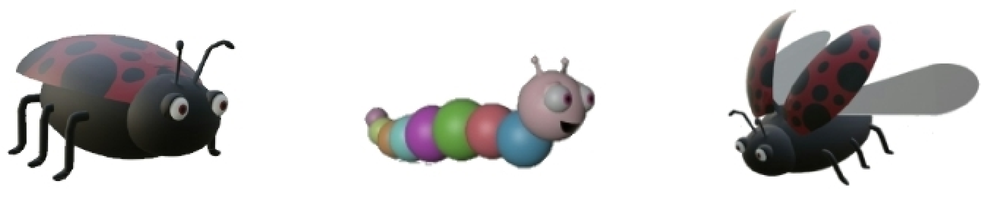

Figure 1.4: Bugs

In addition to traps, the game also features bugs as an obstacle (as shown in Figure 1.4). These bugs typically appear in the second tunnel and are designed to rotate around the tunnel toward the player's position, making them more challenging to evade. However, they can still be avoided by the player. If the player chooses to engage with the bugs, they can be defeated by shooting three bullets at them. If the player collides with a bug, Hans will lose 25% of his battery life (for more information on battery life, see Section 1.4).

1.3.3 Viruses

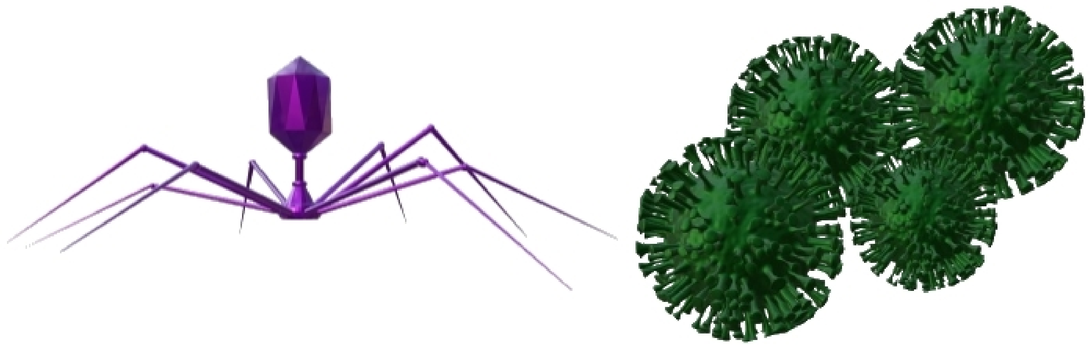

Figure 1.5: Bacteriophage and Rotavirus

The third and final type of obstacle in the game are viruses (illustrated in Figure 1.5). These viruses are typically found in the third tunnel and, similar to bugs, can be eliminated through the use of three bullets. They also, just like bugs, rotate around the tunnel toward the player's position. Bacteriophage, a subtype of virus, will result in an instant death if the player comes into contact with them. Rotaviruses, on the other hand, will cause the player's character to become sick for a brief period of time. During this illness, it is crucial for the player to avoid coming into contact with another Rotavirus, as this will result in the end of the game.

1.4 Additional Features

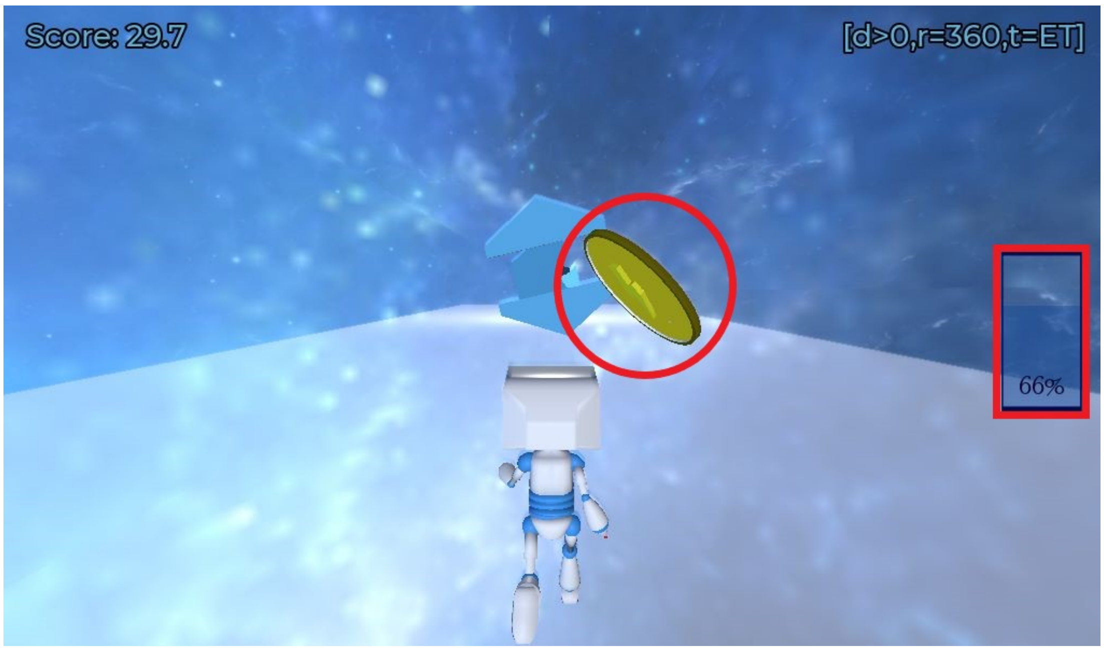

Figure 1.6: Battery and Energy Token

There are several other features of the game that are worth mentioning. One of the most significant of these is the battery life of the player's character, Hans, which is displayed on the right side of the screen (as shown in Figure 1.6). As Hans is designed to resemble a computer, it is necessary for him to recharge his battery throughout the game by collecting energy tokens (Figure 1.6). This will fully restore his battery capacity. There are three main ways in which Hans can lose battery life: running causes a constant reduction of 1%, each bullet shot costs 1% of the battery life, and coming into contact with a bug results in a reduction of 25% (as described in Section 1.3.2). If the battery reaches 0%, Hans will die and the game will end.

Finally, it should be noted that upon successfully navigating through all three types of tunnels, the game will increase in speed and the player will once again encounter the same tunnels, looping through them indefinitely until the player loses.

1.5 Score Count and Winning

The score of the game is based on the length of time that the player is able to survive. Additionally, each time a player successfully shoots down a bug or virus, their score increases by 10 points. As previously mentioned, the game is designed to be played indefinitely, but for the purpose of this study, we have set the game to be considered won after an agent successfully completes nine tunnels, reaching level 10.

2. Implementation of the Game

“Space-run” was developed using the Godot Engine (version v3.2.3.stable) , an open-source game engine licensed under the MIT license. It is a cross-platform tool that offers a range of features for game development, including a visual scripting language, 2D and 3D graphics support, and a powerful physics engine. The Godot Engine utilizes a node-based architecture, where nodes are organized within scenes that can be reused, instanced, inherited, and nested. This structure allows for efficient project management and development within the engine. The game was written entirely in GDScript, the primary scripting language of the Godot Engine. Overall, the Godot Engine provides a user-friendly environment for game development and offers a comprehensive set of features for creating immersive and engaging games.

In addition to using the Godot Engine, the development team also utilized Blender (version 6.2.0) for creating and animating the characters in the game. Blender is a popular open-source 3D modeling and animation software that offers a range of features for creating detailed and realistic characters. The characters were then imported into the Godot Engine using the .glTF 2.0 file format, which is a widely supported file format for exchanging 3D graphics data.

2.1 The top-level organization

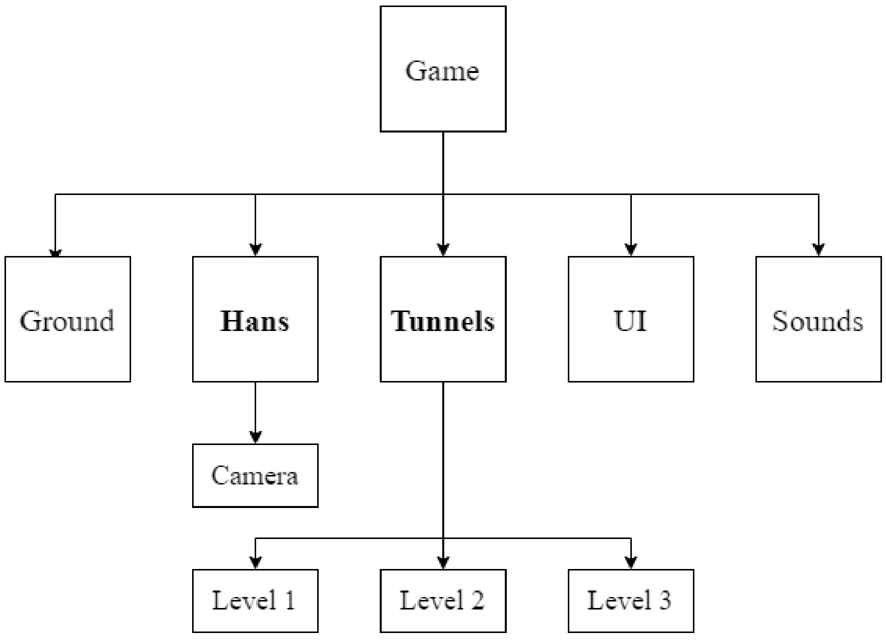

Figure 2.1: Structure of Game.tscn

The main scene for the game, referred to as `Game.tscn`, is depicted in Figure 2.1. It includes several nodes, including Ground, UI, Sound, Game, Hans, and Tunnels. The Ground node is a CSGBox that serves as the ground in the game, while the UI and Sound nodes handle the user interface and audio aspects, respectively. The Game, Hans, and Tunnels nodes contain the majority of the game's functionality. Specifically, the Game node manages the overall gameplay, the Hans node controls the player character, and the Tunnels node manages the movement and appearance of the tunnels.

For a more in-depth understanding, let us examine some of the core aspects of the game in the following sections.

2.2 Game

The script for the Game node is the initial point of the game session and includes both the `_start()` and `_game_over()` methods. It also serves as a link between the game and the agent environment described in Chapter 3, and as such includes all of the necessary set methods for the agent environment. These methods allow for communication between the game and the agent environment, enabling the agent to interact with and influence the game.

The following examples demonstrate the simplified code of the main functions within `Game.gd`. It is important to note that code that is not relevant to any agent except the Keyboard agent, which is the player agent and the only one not designated as a "self-playing agent," has been omitted (and will be in future examples) as it is not relevant to the focus of this thesis. The Keyboard agent allows the player to interact with and control the game, while the other agents, referred to as "self-playing agents," operate independently within the game environment.

```
func _ready():
    # lines relevant to the Keyboard agent:
        ...
    hans.set_speed(num_of_speed_ups)
    hans.set_tunnel_vars(starting_level, tunnel)
    _start()
```

The ready function is called at the start of the game's execution and, after setting up the environment, it triggers the start function. This function initiates the gameplay and sets the necessary conditions for the game to proceed.

```
func _start():
    # add all of obstacle objects specified in the environment
    for name in traps:
        tunnels.scenes["trap_scenes"].append(load("res://Scenes/Traps/"
            + name + ".tscn"))

    for name in bugs:
        tunnels.scenes["bug_scenes"].append(load("res://Scenes/Characters/Bugs/"
            + name + ".tscn"))

    for name in viruses:
        tunnels.scenes["virus_scenes"].append(load("res://Scenes/Characters/Viruses/"
            + name + ".tscn"))

    for name in tokens:
        tunnels.scenes["token_scenes"].append(load("res://Scenes/Tokens/"
            + name + ".tscn"))

    # set Hans's position in front of the chosen tunnel
    match tunnel:
        hans.lvl.TWO:
            hans.translation = Vector3(1250,-32,0)
        hans.lvl.THREE:
            hans.translation = Vector3(-1250,-32,0)

    # add traps to the first tunnel Hans is passing through
    tunnels.create_first_level_traps(tunnel)
```

As described in more detail in Chapter 3, the user can specify environment parameters and a starting level for the agent through the command line. These parameters determine the obstacles that the player will face and the starting position of the player character, Hans. The start function incorporates these parameters into the obstacle arrays and positions Hans accordingly. The function also generates the obstacles for the designated starting level. The creation and deletion of obstacles during gameplay is discussed in Section 2.4 of this chapter.

```
func _game_over():
    if DEBUG:
            # print debugging statement
        print('died on level %d, rot = %d' % [level.level,
            state.state[1]])
    if self_playing_agent:
            # if this is an agent playing, emit the necesarry signal
                that the game is over
        emit_signal("game_finished", score.get_score(), num_of_ticks,
            level.get_level() > max_tunnels, OS.get_ticks_msec() /
            1000.0)
        queue_free()
    else:
            # otherwise handle the game over for the player
```

The game over function manages the end of the game and sends a signal to the top-level script, `Main.gd` (described in Chapter 3), indicating that the game has ended. It also provides `Main.gd` with the necessary information about the game's status and outcome.

2.3 Hans

The next node we want to examine is Hans. While `Hans.tscn` is a scene with the main character and its necesarry animations, what interests us more is the Hans.gd and its key components.

```
func _physics_process(delta):
    # code used to update the variables and UI
    ...

tunnels.delete_obstacle_until_x(curr_tunnel,translation.x -
    tunnels_children[curr_tunnel].translation.x + 50)

# create a trap in the next tunnel every 50 meters
if translation.x < new_trap:
    create_new_trap()

# updating score
score._on_Meter_Passed()

# Hans's movement
var velocity = Vector3.LEFT * speed
velocity = move_and_slide(velocity)

# in case Hans colided with somehting, handle it properly
check_collisions()

# bugs and viruses need to move torwards Hans
tunnels.bug_virus_movement(delta, curr_tunnel)

if isShootingButtonPressed:
    shoot()

# type needs to be calculated before
# so we know where the next trap is on x axis
var type = calc_type()
state.update_state(calc_dist(),calc_rot(),type)
```

The primary function within `Hans.gd` is the `_physics_process()`, which is called on every tick of the game. It handles the main aspects of the player character through the use of various methods and functions. These include deleting passed obstacles, creating new obstacles every 50 meters, updating the score, handling the movement of the player character, bugs and viruses, and determining the current state of the player. The state label, which is displayed on the upper right corner of the screen (as shown in Figure 2.2), is the primary information that agents receive when making decisions about their next move, as described in Chapter 3. The `_physics_process()` function also handles collisions and shooting if the player chooses to do so. Overall, this function plays a crucial role in the gameplay and management of the player character.

It is also worth noting that this script handles the movement of the tunnels to the back as Hans passes them, with the first tunnel being moved to be after the third one. This feature allows for the game to be infinite, as the tunnels are constantly cycled and reused.

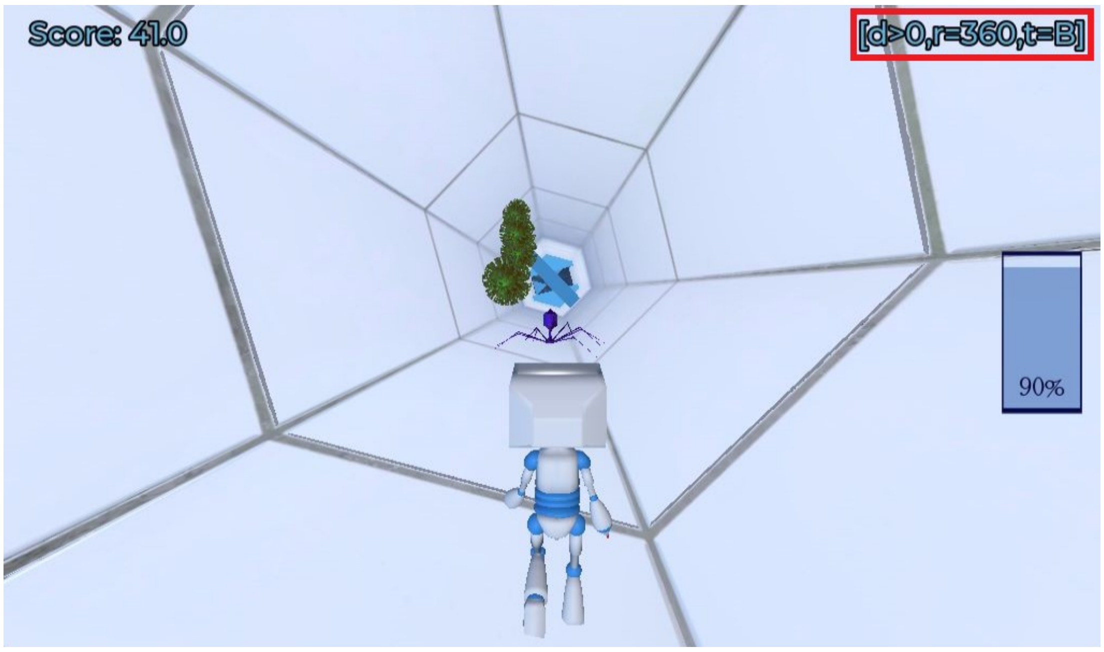

Figure 2.2: State

2.4 Tunnels

The Tunnels node, which is a child of the main scene in the game tree, contains three child nodes of the Spatial type (level1, level2, and level3) and each of these nodes includes a CSGTorus node, which represents the physical appearance of the tunnels. Obstacles are added to the appropriate level node as instances. The `Tunnels.gd` script, which is attached to the Tunnels node, handles many of the previously mentioned functions such as obstacle creation and deletion and tunnel rotation. In the following code snippets, we will examine the Tunnels.gd script in greater detail.

```
func _physics_process(delta):
    # gets move from the agent
    # in case we chose the Keyboard agent
    #this will just return input from the keyboard
    var move = game.agent.move(game.state.get_state(),
       game.score.get_score(), game.num_of_ticks)

    #rotates the tunnel
    if move[0] == 1:
       var tunnel = get_child(hans.get_current_tunnel())
       tunnel.rotate_object_local(Vector3.RIGHT,-ROTATE_SPEED * delta)
    elif move[0] == -1:
       var tunnel = get_child(hans.get_current_tunnel())
       tunnel.rotate_object_local(Vector3.LEFT,-ROTATE_SPEED * delta)

    # shoot if necesarry
    if not hans == null: # if it is not instanced we can't call the
       function
       hans.switch_animation(move[1] == 1)
```

The _physics_process() function within the Tunnels.gd script serves as the primary connection between the agent and the game. As shown in the provided code, the function retrieves the next move from the agent and rotates the tunnel accordingly, potentially including shooting as well.

```
func create_first_level_traps(tunnel):
    # get the level we are making traps for
    var level = tunnel

    # pick number of traps to be added
    var num_of_traps = rand.randi_range(50,65)
    var x = 1200

    for n in num_of_traps:
        # add space between traps
        x -= rand.randi_range(TRAP_RANGE_FROM,TRAP_RANGE_TO)
        # check if the trap will be inside the tunnel
        # if not, break
        if x < -1200:
            break
        create_one_obstacle(level, x)
```

The function depicted in the code above serves to generate obstacles in the starting tunnel. By periodically creating traps in the tunnel ahead, the game is able to prevent lag caused by an excessive number of objects existing simultaneously. For that reason, this function is used only once, at the beginning of the game.

```
func create_one_obstacle(level,x):
    # pick which kind of obstacle will be added
    # this process is partially random, depending on weather the tunnel
    # can contain one or more kinds of obstacles
    var scene = pick_scene(level)

    # get the level we are making traps for
    var tunnel = get_child(level)

    # randomly pick an obstacle
    var i = pick_obstacle(scene)

    # make an instance
    var obstacle = scene[i].instance()
    obstacle.translation.x = x

    tunnel.add_child(obstacle)

    # randomly pick obstacle rotation
    rotate_obstacle(obstacle)
```

The tunnels are positioned along the x axis, and this function allows for the creation of obstacles within them at specific x positions.

```
func delete_obstacle_until_x(level,x):
    var tunnel = get_child(level)
    # loop through children of the tunnel
    # and eliminate any obstacles
    # until position x
    for obstacle in tunnel.get_children():
        if not "light" in obstacle.name and not "torus" in
            obstacle.name and not "Bullet" in obstacle.name:
            if obstacle.translation.x > x:
                obstacle.queue_free()
            else:
                return
```

As previously mentioned, by dynamically deleting passed obstacles, the game is able to maintain a stable performance and avoid overloading the system. The provided code demonstrates the implementation of this function.

3. Structure of the Experimental Setting

To train the agent on a specific environment, the user must utilize a command line interface. There are various options available to cater to the user's needs. The following sections will outline all of the provided options and how they are encoded.

3.1 Command line options

n=int	number of games
agent=string	name of the agent
level=int	number of the level to start from
env=[string]	list of obstacles that will be chosen in the game
shooting=string	enable or disable shooting
dists=int	number of states in a 100-meter interval
rots=int	number of states in 360 degrees rotation
database=string	read the data for this command from an existing file and/or update the data after the command is executed
ceval=bool	performs continuous evaluation
debug=bool	display debug print statements
options	displays options

Note: any of these options can be omitted as they all have default values. If no options are specified, a normal game with the Keyboard agent will be run.

3.1.1 Command line option descriptions

n - Number of games the agent will train on in this session

agent - Name of the desired agent
Options: [Keyboard, Static, Random, MonteCarlo, SARSA, QLearning, ExpectedSARSA, DoubleQLearning]
Sub-options (only for the listed agents): MonteCarlo, SARSA, QLearning, ExpectedSARSA, DoubleQLearning =[float, float, float, float] : [gam (range [0,1]), eps (range [0,1]), epsFinal (range [0,1]), initOptVal [0,)]
Example usage: "MonteCarlo:eps=0.1,gam=0.2"

level - Number of the level to start from
Options: [1, ... , 10]
Note: after the 10th level, the agent is considered to have won the game

env - List of obstacles that will be chosen in the game
Options (any subset of): [Traps, Bugs, Viruses, Tokens, I, O, MovingI, X, Walls, Hex, HexO, Balls, Triangles, HalfHex, Worm, LadybugFlying, LadybugWalking, Rotavirus, Bacteriophage]

Note: if this parameter is not included, the environment will contain all available options (i.e. the full game).

shooting - Enable or disable shooting
Options: [enabled, disabled]
Note: this option is disabled by default.

dists - Number of states in a 100-meter interval
This parameter is part of the state label and typical options range from 1 to 3. Default value is 1.

rots - Number of states in 360 degrees rotation
This parameter is part of the state label and the minimum viable option is 6. This is also the default value.

database - Read or write data for this command from/to a file
Options: [read, write, read_write]
Note: these files are typically used to start another session of the agent's training from the last point of the previous session, to run a game with visuals and observe the agent's performance, or for plotting the results. This option does not affect the Keyboard, Static, or Random agents.

ceval - Performs continuous evaluation
This parameter, with a value of true or false, indicates that after each training game, a test game will be played using only the policy(s) learned thus far. For example, if the user specifies "n=100", a total of 200 games will be executed, with 100 of them being training games and the remaining 100 being test games. This allows for the assessment of the agent's progress and performance during the training process.

debug - Display debug print statements
Options: [true,false]

options - Displays all of the mentioned options

3.2 Main

The`Main.tscn` scene is the top level scene in the game and consists of a single Node type node. The script attached to this node, Main.gd, is responsible for ensuring that all options specified in the command line (as discussed in Section 3.1) are executed correctly. This script is the starting point of the training and handles the initialization and execution of ore or more game sessions. There are several key functions within the `Main.gd` script that are worth discussing in more detail.

```
func _ready():
    # get args
    var unparsed_args = OS.get_cmdline_args()
    # show options
    if unparsed_args.size() == 1 and unparsed_args[0] == "options":
        display_options()

    # parse agrs
            ...

    # set param, if something went wrong, show options
    if set_param(args) == false:
        display_options()
    else:
        # make an instance of the chosen agent
        instance_agent()
        # this filename will contain all options
        # and will use as a unique key for the training session
        build_filename()
        # there was a problem while initializing an agent
        if not agent_inst.init(actions, read, write, command, n, debug):
            print("Something went wrong, please try again")
            print(options)

        play_game()
```

The _ready() function is the starting point of the program when run from the command line. It is responsible for parsing all of the arguments and checking their validity. If any issues are encountered, the program will display options and terminate. If the arguments are valid, the first game will be played.

```
func play_game():
    if agent == "Keyboard" and VisualServer.render_loop_enabled: # play
        a regular game
       ...
    # there are still some number of games that need to be played
    elif n > 0:
        n -= 1
        game = game_scene.instance()
        set_param_in_game()
        # prepare the agent
        agent_inst.start_game(is_eval_game)
    # we came to the end of the session
    else:
        agent_inst.save(write)
        print_and_write_ending()
```

The play_game() function is called each time a game is played. If the specified number of games (as defined by the ”n” parameter) have already been played, the program will terminate. Otherwise, a single game will be executed.

```
func on_game_finished(score, ticks, win, time):
    # finish up
    print_and_write_score(score, win)
    agent_inst.end_game(score, time)
    # start a new game
    play_game()
```

The `game_over()` function is called when the game emits a signal indicating that it has finished. Upon execution, this function outputs the necessary information, updates the agent through the `end_game()` function, and then calls the `play_game()` function to continue the game session.

4. Applied Algorithms

talk ab algos

4.1 Title of the first subchapter of the second chapter

4.2 Title of the second subchapter of the second chapter

5. Implementation of the Agents

code structure of the agents

5.1 Title of the first subchapter of the second chapter

5.2 Title of the second subchapter of the second chapter

6. Experiments

6.1 Title of the first subchapter of the second chapter

6.2 Title of the second subchapter of the second chapter

Conclusion

Bibliography

Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction.* MIT press, Praha, 2018.

List of Figures

List of Tables

List of Abbreviations

A. Attachments

A.1 First Attachment