

VizDoom-Installationsanleitung für Windows unter Nutzung von WSL2

Sean Mulready

2024-01-10

Diese Anleitung soll dabei helfen VizDoom auf Windows zu installieren, was bisher nur schwierig bis unmöglich war und letztlich 3 Szenarien spielen und verschiedene Daten dazu aufzeichnen können. Um VizDoom ohne Probleme auf Windows installiert zu bekommen bedienen wir uns eines einfachen Tricks: Wir führen das Programm auf einer Plattform aus, welche VizDoom problemlos installieren und ausführen kann: Linux. Windows bietet uns dazu das Windows Subsystem für Linux (kurz WSL) an. Zunächst geht es also um die Installation von WSL, anschließend der Installation und Konfiguration von VizDoom. Schlussendlich ein Script um das Spiel aktiv spielen zu können und Daten zu erfassen.

1 Installation, Nutzung und der Code erklärt

1.1 WSL installieren

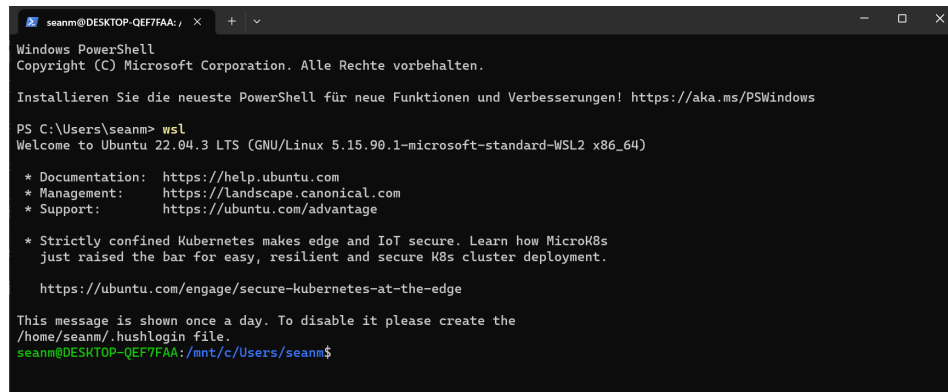
Um VizDoom auf Linux auszuführen, brauchen wir logischerweise Linux. Die Installation geht recht einfach:

- Zunächst über das Windows-Menü die Powershell (im Administratormodus) öffnen
- Sofern WSL noch nicht installiert ist genügt der Befehl “wsl –install”
- Per Default wird WSL2 installiert, welches im Gegensatz zu WSL1 GUI von Linux-Apps unterstützt
- Die Version von Linux kann manuell gewählt werden, standardmäßig wird Ubuntu installiert
- Nähere Informationen zur Installation und auch der manuellen Wahl der Linux-Distribution gibt es [hier](#)
- Im Zuge der Installation muss ein User angelegt und ein Passwort vergeben werden.

1.2 WSL starten, updaten ,Python, Pip-installer, Vizdoom installieren

Standardmäßig ist Python auf Ubuntu vorinstalliert, aber es schadet grundsätzlich nicht, das Linux-System und seine Komponenten auf den neusten Stand zu bringen.

- Zunächst muss WSL nun aber gestartet werden. Das geht denkbar einfach: In die Windows Powershell wird “wsl” eingegeben und Enter gedrückt. Damit startet das Subsystem. Alle weiteren Befehle, also auch das starten von Apps, Prozessen und Programmen wird hier per Befehl gestartet.



```
seanm@DESKTOP-QEF7FAA: /mnt/c/Users/seanm$ wsl
Windows PowerShell
Copyright (C) Microsoft Corporation. Alle Rechte vorbehalten.

Installieren Sie die neueste PowerShell für neue Funktionen und Verbesserungen! https://aka.ms/PSWindows

PS C:\Users\seanm> wsl
Welcome to Ubuntu 22.04.3 LTS (GNU/Linux 5.15.90.1-microsoft-standard-WSL2 x86_64)

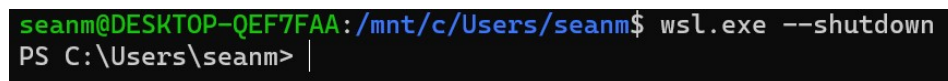
 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

 * Strictly confined Kubernetes makes edge and IoT secure. Learn how MicroK8s
   just raised the bar for easy, resilient and secure K8s cluster deployment.
   https://ubuntu.com/engage/secure-kubernetes-at-the-edge

This message is shown once a day. To disable it please create the
/home/seanm/.hushlogin file.
seanm@DESKTOP-QEF7FAA: /mnt/c/Users/seanm$
```

Abbildung 1. WSL starten.

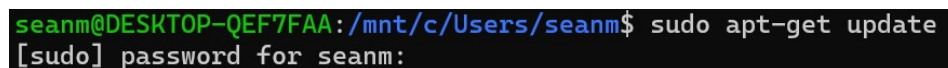
- Um WSL zu schließen genügt der Befehl “wsl.exe –shutdown”



```
seanm@DESKTOP-QEF7FAA: /mnt/c/Users/seanm$ wsl.exe --shutdown
PS C:\Users\seanm>
```

Abbildung 2. WSL schließen

- Linux auf den neusten Stand bringen: je nach Version ist es entweder der “apt-get” oder der “apt” Befehl. Im weiteren Verlauf dieses Guides wird der “apt-get” Befehl verwendet. Um im gestarteten Subsystem nach Updates zu suchen wird zunächst der Befehl “sudo apt-get update” und anschließend der Befehl “sudo apt-get upgrade”. Beim ersten “sudo”-Befehl gibt es eine Passwort-Abfrage, das Passwort wurde bei der Benutzererstellung vergeben. Vielleicht ungewohnt für Menschen die nicht mit Linux vertraut sind: Bei der Passworтеingabe erscheinen keine Zeichen, keine Punkte, auch der Cursor bewegt sich nicht. Passwort eingeben, Enter drücken, fertig.



```
seanm@DESKTOP-QEF7FAA: /mnt/c/Users/seanm$ sudo apt-get update
[sudo] password for seanm:
```

Abbildung 3. Updates suchen und Passworтеingabe

```
seanm@DESKTOP-QEF7FAA:/mnt/c/Users/seanm$ sudo apt-get upgrade
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
Calculating upgrade... Done
The following packages have been kept back:
 alsa-ucm-conf
The following packages will be upgraded:
  bind9-dnswilds bind9-host bind9-libs libc-bin libc-dev-bin libc-devtools libc6 libc6-dev libcups2 libvpx7 libx11-6
  libx11-data libx11-dev libx11-xcb1 libxpm4 linux-libc-dev locales ubuntu-advantage-tools
18 upgraded, 0 newly installed, 0 to remove and 1 not upgraded.
1 not fully installed or removed.
Need to get 14.5 MB/16.2 MB of archives.
After this operation, 104 kB of additional disk space will be used.
Do you want to continue? [Y/n] Y
```

Abbildung 4. Updates durchführen

- PIP installieren: PIP wird einfach über den Befehl “sudo apt-get install python3-pip -y” installiert

```
seanm@DESKTOP-QEF7FAA:/mnt/c/Users/seanm$ sudo apt-get install python3-pip -y
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
python3-pip is already the newest version (22.0.2+dfsg-1ubuntu0.3).
0 upgraded, 0 newly installed, 0 to remove and 1 not upgraded.
seanm@DESKTOP-QEF7FAA:/mnt/c/Users/seanm$
```

Abbildung 5. PIP installieren

- VizDoom installieren: Jetzt wird es mehr oder minder spannend, aber auch am zeitintensivsten. Wir installieren Vizdoom jetzt einfach via “pip install vizdoom”. Enter drücken, zurücklehnen und den Computer den Rest machen lassen.

```
seanm@DESKTOP-QEF7FAA:/mnt/c/Users/seanm$ pip install vizdoom
```

Abbildung 6. VizDoom installieren

- ViZDoom updaten: Um ViZDoom hin und wieder auf den neusten Stand zu bringen, lohnt es sich, den Update-Befehl auszuführen. Der lautet einfach: “pip install –upgrade vizdoom”

1.3 Die Programmieroberfläche, ihre Erweiterungen und Ordnerstrukturen anlegen

Als Programmieroberfläche bietet sich Visual Studio Code an. Es gibt 3 Extensions (Erweiterungen), die heruntergeladen werden müssen:

- WSL
- Python
- R

Die Extension WSL sorgt dafür, dass wir WSL in Visual Studio Code nutzen können, die Extensions für Python und R wiederum machen die entsprechenden Skripte ausführbar. Extensions installiert man in VS Code, indem man entweder auf den Button dafür geht oder die Tastenkombination STRG+SHIFT+x drückt und anschließend über das Suchfeld nach den entsprechenden Erweiterungen sucht und sie installiert.



Abbildung 7. Button für die Extensions (Mitte)

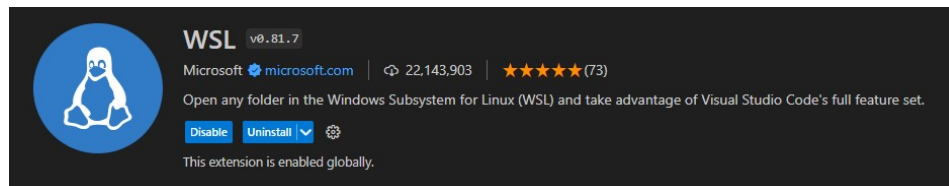


Abbildung 8. WSL-Extension

Der einfachste Weg um WSL in VS Code zu starten: Bei geöffnetem VSC F1 drücken, “wsl” im Suchfeld eingeben und, wenn sonst nichts in VSC gemacht werden soll “Connect to WSL” wählen. Meiner Meinung nach besser ist es aber, WSL in einem neuen Fenster via “Connect WSL in a new Window”, also vor allem wenn gerade beispielsweise ein Quarto-Dokument, welches über Windows läuft, noch offen ist. Gut erkennbar, dass das Starten geklappt hat ist das Suchfeld, welches nicht nur WSL sondern auch die genutzte Distribution (hier: Ubuntu) anzeigt.

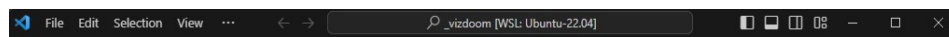


Abbildung 9. WSL läuft

Um die Ordnerstrukturen anzulegen eignet sich der ganz gewöhnliche File-Explorer von Windows. Wer diesen öffnet wird bei genauerem Hinsehen bemerken, dass ein neues Verzeichnis mit dem Namen “Linux” aufgetaucht ist. Innerhalb von Linux befindet sich der Ordner mit dem Namen der Linux-Distribution. Theoretisch ist man jetzt frei darin wo man nachfolgend die Ordner anlegt, aber ich finde “home/user/” (anstelle von user steht der vergebene Nutzename) ist ein guter Anlaufpunkt.

Es sollten direkt zwei Ordner angelegt werden. Einen für die Scripte und Daten, und ein Ordner nur für die Konfigurationsdatei um die Tastenbelegung selbst anpassen zu können, dazu später mehr. Ich habe mich für einen Ordner mit dem Titel “vizdoom” und einen weiteren mit dem Namen “vizdoom_config” entschieden

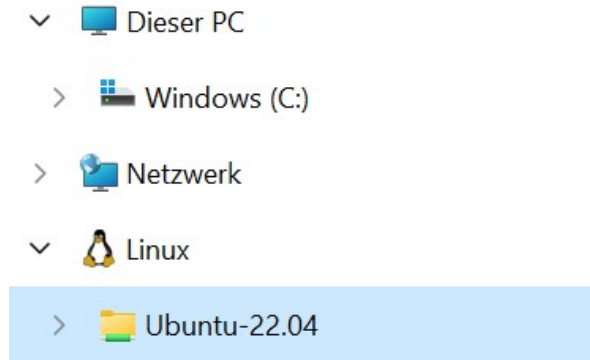


Abbildung 10. Das neue Verzeichnis

1.4 Die Steuerungskonfiguration

Standardmäßig nutzt ViZDoom die Tastenbelegung von ZDoom, welche jedoch stellenweise stutzig macht und nicht sehr intuitiv erscheint. So geht man nach links durch “,” und nach rechts durch “.”. Also warum nicht ändern? Und das lässt sich am besten dadurch gestalten, dass man die “_vizdoom.ini”-Datei entsprechend umschreibt.

Normalerweise wird diese Datei automatisch beim ersten Durchlauf erstellt, jedoch sucht man sie vergeblich, auch nach einem ersten Durchlauf, wenn man den hier beschriebenen Schritten gefolgt ist. Zum Glück gibt es ein workaround: Man besorge sich die _vizdoom.ini Datei [hier](#) (zur Not einfach kopieren, den Texteditor öffnen, einfügen und dann “speichern unter”) und lege sie im dafür vorgesehenen Ordner ab - bei mir “vizdoom_config”.

Den nächsten Schritt kann - wer die Datei kopiert und dann im Text-Editor eingefügt hat - auch vor dem Speichern ausführen: Zur Sektion [Doom.Bindings] navigieren (recht weit unten) und dort durch die Befehle gehen. Uns interessieren

- “moveleft/moveright” (was eine seitliche Bewegung mit Blick nach vorne bedeutet)
- “left/right” (entspricht einer Drehung in die jeweilige Richtung)
- “forward/back” (entspricht vorwärts und rückwärts)

Es gibt nun zwei Möglichkeiten: Erstens, man ändert die entsprechenden Zeichen der Zeile einfach um und speichert. Zweitens, man fügt eine weitere Zeile hinzu. Wichtig: das Ganze muss, wenn es als zusätzliche Taste programmiert wird entsprechend geschrieben werden nach dem Schema: [Taste]=+[Befehl]. Letzterer Fall ist in Abbildung 11 für das seitliche Bewegen gezeigt. in meinem Falle habe ich letztlich die “klassische” wasd-Steuerung gewählt. Für das seitliche gehen nach links und rechts habe ich “q” und “e” programmiert. Es ist zu überlegen, welche Bewegungen in welchem Szenario gemacht werden können und entweder entsprechende “_vizdoom.ini”-Dateien zu schreiben oder einzelne Tasten doppelt zu belegen, sofern sich die zugehörigen Bewegungen nicht überschneiden. Hier seien den eigenen Vorlieben keine bzw. kaum Grenzen gesetzt.

```
,+=moveleft  
.=+=moveright  
q+=moveleft  
e+=moveright
```

Abbildung 11. Tastenbelegung: zusätzlich zu “,” und “.” können jetzt auch “q” und “e” für seitliche Bewegungen verwendet werden

1.5 Das Script

Jetzt geht es los. Wir öffnen Visual Studio Code, gehen zu WSL. Über die Navigation links können wir zu unserem erstellten Ordner (hier: “_vizdoom”) gehen, bzw. diesen Ordner öffnen. Wir erstellen über den Reiter “File” oben links eine neue Datei. VS Code fragt automatisch nach dem Format und wir wählen “Python” (siehe Abbildung 12).

Ab hier werden wir uns vorerst nur noch mit dem Basic-Szenario beschäftigen, Modifikationen für andere Szenarien sind für die Zukunft geplant.

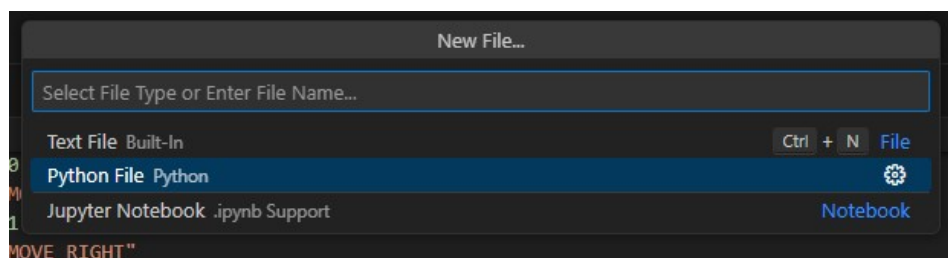


Abbildung 12. Das Python-Format wählen

Zunächst nennt sich diese Datei “Untitled-1”. Durch drücken von STRG+s wird die Datei gespeichert. Es öffnet sich ein neues Fenster und wir können den Dateinamen vergeben. Da ich hier mit dem Spectator-Modus arbeite, nenne ich sie “Spectator”.

Nachfolgend einmal der Code in Gänze, bevor die zu modifizierenden Zeilen gesondert hervorgehoben werden:

```
from __future__ import print_function
import vizdoom as vzd
import csv
import time
import numpy as np

# Enter Subject Data
sub_id = "01"

# define a number of blocks to write a file after each block so in case
# the game crashes for any reason, not all data is lost
blocks = 2
for b in range(blocks):

    game = vzd.DoomGame()
    # where to get the .ini-file from
    game.set_doom_config_path("/home/seanm/vizdoom_config/_vizdoom.ini")

    # Choose the scenario config file you wish to watch.
    # Don't load two configs because the second will overwrite the first one.
    # Multiple config files are okay, but combining these ones doesn't make much sense.

    game.load_config("/home/seanm/.local/lib/python3.10/site-packages/vizdoom/scenarios/basic.cfg")

    # Enables information about all objects present in the current episode/level.
    game.set_objects_info_enabled(True)

    # Enables information about all sectors (map layout).
    game.set_sectors_info_enabled(True)

    # Clear all game variables first to unify the variables for all scenarios
    game.clear_available_game_variables()

    # Add game variables for Health and Ammo
    game.add_available_game_variable(vzd.GameVariable.HEALTH)
    game.add_available_game_variable(vzd.GameVariable.AMMO2)

    # Add Game Variables for the position
    pos_x = game.add_available_game_variable(vzd.GameVariable.POSITION_X)
    pos_y = game.add_available_game_variable(vzd.GameVariable.POSITION_Y)
    pos_z = game.add_available_game_variable(vzd.GameVariable.POSITION_Z)
    angle = game.add_available_game_variable(vzd.GameVariable.ANGLE)

    # Set screen size
    game.set_screen_resolution(vzd.ScreenResolution.RES_1280X960)

    # Enables spectator mode so you can play, but your agent is supposed to watch, not you.
    game.set_window_visible(True)
    game.set_mode(vzd.Mode.SPECTATOR)
    game.init()

    # Define a translation function for the last action
    def translate_action(last_action):

        last_action_trnsl = "NA"
        #scenario basic
```

```

        if last_action == [1.0,0.0,0.0]:
            last_action_trnsl = "MOVE_LEFT"
        elif last_action == [0.0,1.0,0.0]:
            last_action_trnsl = "MOVE_RIGHT"
        elif last_action == [0.0,0.0,1.0]:
            last_action_trnsl = "ATTACK"

    return last_action_trnsl

#function to return objects info
def find_object_data(object_list,object_id,object_name):
    for o in object_list:
        if o.id == object_id and o.name != object_name:
            return o
    #return None

# Specify how many Episodes
episodes = 10

# header for the csv-file
columns = ["Episode",
           "State",
           "Tic",
           "Health",
           "Ammo",
           "x_pos",
           "y_pos",
           "z_pos",
           "angle/orientation",
           "Action",
           "Reward",
           "Cumulative_Reward",
           "Time",
           "Object_ID",
           "Object_Name",
           "Object_PosX",
           "Object_PosY",
           "Object_PosZ",
           "Object_Angle",
           "App_Object_ID",
           "App_Object_Name",
           "App_Object_PosX",
           "App_Object_PosY",
           "App_Object_PosZ",
           "App_Object_Angle"
          ]

# Open a CSV file to store episode-data of each block

block_filename = sub_id + "_game_data_block_"+ str(b+1) + ".csv"
with open(block_filename,'w', newline='') as starting_file:
    csv_writer = csv.writer(starting_file)
    csv_writer.writerow(columns)

```



```

#start with columns-header for the Episode_End_Data
columns2 = ["Episode",
            "Total_Reward",
            "Time",
            "FPS"]

# Open another csv-file
endrow_filename = sub_id + "_endrows_block_" + str(b+1) + ".csv"
with open(endrow_filename, 'w', newline='') as end_file:
    csv_writer = csv.writer(end_file)
    csv_writer.writerow(columns2)

#Use a set to keep track of uniqueobject IDs
unique_object_ids = set()

#create single column arrays for each variable to later concatenate into a dataframe
Episode_arr = np.full((episodes*151,1), np.nan, dtype = np.int32)
State_arr = np.full((episodes*151,1), np.nan, dtype = np.int32)
Tic_arr = np.full((episodes*151,1), np.nan, dtype = np.int32)
Health_arr = np.full((episodes*151,1), np.nan, dtype = np.int32)
Ammo_arr = np.full((episodes*151,1), np.nan, dtype = np.int32)
xpos_arr = np.full((episodes*151,1), np.nan, dtype = np.float64)
ypos_arr = np.full((episodes*151,1), np.nan, dtype = np.float64)
zpos_arr = np.full((episodes*151,1), np.nan, dtype = np.float64)
angle_arr = np.full((episodes*151,1), np.nan, dtype = np.float64)
Action_arr = np.full((episodes*151,1), np.nan, dtype = np.dtype('U10'))
Reward_arr = np.full((episodes*151,1), np.nan, dtype = np.int32)
CumReward_arr = np.full((episodes*151,1), np.nan, dtype = np.int32)
Time_arr = np.full((episodes*151,1), np.nan, dtype = np.float64)
ObjID_arr = np.full((episodes*151,1), np.nan, dtype = np.int32)
Objname_arr = np.full((episodes*151,1), np.nan, dtype = np.dtype('U10'))
Objx_arr = np.full((episodes*151,1), np.nan, dtype = np.float64)
Objy_arr = np.full((episodes*151,1), np.nan, dtype = np.float64)
Objz_arr = np.full((episodes*151,1), np.nan, dtype = np.float64)
Objang_arr = np.full((episodes*151,1), np.nan, dtype = np.float64)
AppobjID_arr = np.full((episodes*151,1), np.nan, dtype = np.int32)
Appobjname_arr = np.full((episodes*151,1), np.nan, dtype = np.dtype('U12'))
Appobjx_arr = np.full((episodes*151,1), np.nan, dtype = np.float64)
Appobjy_arr = np.full((episodes*151,1), np.nan, dtype = np.float64)
Appobjz_arr = np.full((episodes*151,1), np.nan, dtype = np.float64)
Appobjangle_arr = np.full((episodes*151,1), np.nan, dtype = np.float64)

# Creating arrays for the Episode_End_Data

End_Eps_arr = np.full((episodes,1), np.nan, dtype = np.int32)
End_Rew_arr = np.full((episodes,1), np.nan, dtype = np.int32)
End_Time_arr = np.full((episodes,1), np.nan, dtype = np.float64)
End_FPS_arr = np.full((episodes,1), np.nan, dtype = np.float64)

#starting the index
current_index = 0

# Loop through episodes
for i in range(episodes):

    print("Episode #" + str(i + 1))
    game.new_episode()

```

```

cumulative_reward = 0 # Initialize cumulative reward
start_time = time.time() # Record the start time


while not game.is_episode_finished():

    state = game.get_state()
    game.advance_action()
    last_action = game.get_last_action()
    reward = game.get_last_reward()
    last_action_trnsl = translate_action(last_action)

    cumulative_reward += reward # Update cumulative reward
    current_time = time.time() - start_time

    object_ids = []
    for o in state.objects:
        object_ids.append(o.id)
        unique_object_ids.update(object_ids)

    # Collect data for each time step within the episode so for each array
    Episode_arr[current_index] = i+1
    State_arr[current_index] = state.number
    Tic_arr[current_index] = game.get_episode_time()
    Health_arr[current_index] = game.get_game_variable(vzd.GameVariable.HEALTH)
    Ammo_arr[current_index] = game.get_game_variable(vzd.GameVariable.AMMO2)
    xpos_arr[current_index] = game.get_game_variable(vzd.GameVariable.POSITION_X)
    ypos_arr[current_index] = game.get_game_variable(vzd.GameVariable.POSITION_Y)
    zpos_arr[current_index] = game.get_game_variable(vzd.GameVariable.POSITION_Z)
    angle_arr[current_index] = game.get_game_variable(vzd.GameVariable.ANGLE)
    Action_arr[current_index] = last_action_trnsl
    Reward_arr[current_index] = reward
    CumReward_arr[current_index] = cumulative_reward
    Time_arr[current_index] = current_time

    # get data for Cacodemon
    for object_id in unique_object_ids:
        if object_id == 0:
            object_data = find_object_data(state.objects, object_id, "DoomPlayer")
            if object_data is not None:
                # fill corresponding arrays with data of the Cacodemon
                ObjID_arr[current_index] = object_data.id
                Objname_arr[current_index] = object_data.name
                Objx_arr[current_index] = object_data.position_x
                Objy_arr[current_index] = object_data.position_y
                Objz_arr[current_index] = object_data.position_z
                Objang_arr[current_index] = object_data.angle
            else:
                # what data to fill the array with if there is none
                ObjID_arr[current_index] = 0
                Objname_arr[current_index] = "None"
                Objx_arr[current_index] = 0

```

```

        Objy_arr[current_index] = 0
        Objz_arr[current_index] = 0
        Objang_arr[current_index] = 0

# get data for the other objects
for object_id in unique_object_ids:
    if object_id != 0:
        object_data = find_object_data(state.objects, object_id, "DoomPlayer")
        if object_data is not None:
            # fill arrays with the data
            AppobjID_arr[current_index] = object_data.id
            Appobjname_arr[current_index] = object_data.name
            Appobjx_arr[current_index] = object_data.position_x
            Appobjy_arr[current_index] = object_data.position_y
            Appobjz_arr[current_index] = object_data.position_z
            Appobjangle_arr[current_index] = object_data.angle
        else:
            # what data to fill the array with if there is none
            AppobjID_arr[current_index] = 0
            Appobjname_arr[current_index] = "None"
            Appobjx_arr[current_index] = 0
            Appobjy_arr[current_index] = 0
            Appobjz_arr[current_index] = 0
            Appobjangle_arr[current_index] = 0

    current_index += 1 #increment index

# when to stop the episode, default by config-file is 300 tics
if state.number > 150:
    break

End_Eps_arr[i] = i+1
End_Rew_arr[i] = game.get_total_reward()
End_Time_arr[i] = (time.time()-start_time)
End_FPS_arr[i] = (state.number/(time.time()-start_time))

print("Episode finished!")
print("Total reward:", game.get_total_reward())
print("Time:", (time.time() - start_time))
print("*****")
time.sleep(0.5)

game.close()

Episodes_summary = np.concatenate((End_Eps_arr,

```

```

        End_Rew_arr,
        End_Time_arr,
        End_FPS_arr),
        axis = 1)

with open(endrow_filename, 'a', newline='') as summary_file:
    csv_writer = csv.writer(summary_file)
    csv_writer.writerow(Episodes_summary)

all_episode_data = np.concatenate((Episode_arr,
    State_arr,
    Tic_arr,
    Health_arr,
    Ammo_arr,
    xpos_arr,
    ypos_arr,
    zpos_arr,
    angle_arr,
    Action_arr,
    Reward_arr,
    CumReward_arr,
    Time_arr,
    ObjID_arr,
    Objname_arr,
    Objx_arr,
    Objy_arr,
    Objz_arr,
    Objang_arr,
    AppobjID_arr,
    Appobjname_arr,
    Appobjx_arr,
    Appobjy_arr,
    Appobjz_arr,
    Appobjangle_arr),
    axis=1)

# Write episode_data to CSV file after each block
with open(block_filename, 'a', newline='') as file:
    csv_writer = csv.writer(file)
    csv_writer.writerow(all_episode_data)

```

1.6 Das Script anpassen

Folgende Zeilen müssen angepasst werden:

```

#tell where to get the _vizdoom.ini-file from
game.set_doom_config_path("/home/seanm/vizdoom_config/_vizdoom.ini")

```

In dieser Zeile wird dem Programm gesagt, von wo es die Steuerungskonfiguration beziehen soll, die man zuvor nach eigenen Wünschen modifiziert hat.

```

# specify the scenario
game.load_config("/home/seanm/.local/lib/python3.10/site-packages/vizdoom/scenarios/basic.cfg")

```

Hier muss der Pfad geändert werden. In der Regel sollte es derselbe Pfad wie hier sein, abgesehen vom Username (nach /home). Er kann sich aber auch unterscheiden, wenn Vizdoom zum Beispiel unter einer anderen Python-Version installiert wurde.

Hat man diese Änderungen vorgenommen, kann man dem ganzen einen Testlauf geben. Hierzu oben rechts auf den Play-Button drücken. Was dann passieren sollte ist Folgendes: Das Spiel sollte sich öffnen und man hat 10 mal ca. 4 Sekunden Zeit um das Monster abzuschießen (wenn die Konfiguration wie angegeben geändert wurde, nach links oder rechts mit den Tasten Q und E, respektive. Schießen mit der linken Maustaste (Mauszeiger muss im Fenster sein) oder der STRG-Taste). Wenn die 10 mal um sind (sollte man länger als die ca. 4 Sekunden warten geht es automatisch zum nächsten Versuch) sollte sich das Fenster kurz schließen, um sich dann wieder zu öffnen und erneut 10 mal die Möglichkeit zu geben, das Monster zu erschießen. Danach sollten im Ordner in dem das Script liegt auch 4 neue Dateien liegen die wie folgt benannt sind:

- “01_game_data_block_1.csv”
- “01_game_data_block_2.csv”
- “01_endrows_block_1.csv”
- “01_endrows_block_2.csv”

Hat alles so geklappt? Super. Dann beschäftigen wir uns im folgenden Abschnitt damit, warum das alles so passiert.

1.7 Der Code step-by-step

Nachdem nun alles läuft, ein paar Runden im Szenario gespielt wurden, kommt vielleicht die erste Neugier, was dieser ganze Code denn jetzt eigentlich genau macht. Dafür ist dieses Kapitel gedacht. Der Code wird nochmal in einzelnen “Happen” gezeigt und daran entlang erklärt, was hier eigentlich geschieht, zumindest im Bereich des Endnutzers. Einen tieferen Einblick in die Mechanik des Spiels gibt es an dieser Stelle nicht.

Wir beginnen ganz am Anfang und damit beim Importieren:

```
from __future__ import print_function
import vizdoom as vzd
import csv
import time
import os
import pandas as pd
```

Wir beginnen damit, alles mehr oder minder notwendige zu importieren. Mit der “import”-Funktion lädt man sogenannte “libraries” also Bibliotheken. Man kann es sich tatsächlich in der Funktion wie eine Bibliothek vorstellen: Man holt sich ganz viel verfügbares Wissen in

einen “Raum”, auf den man bei Bedarf zugreifen kann. Wie ich darauf zugreife kommt noch später in diesem Abschnitt, jedoch seien hier zwei Funktionalitäten erwähnt:

- “**import..as..**”: Wenn ich später Wissen aus einer meiner Bibliotheken verwenden möchte muss ich dem Programm jedes mal sagen, aus welche Bibliothek es das Wissen beziehen soll. Um Zeit zu sparen und der Übersicht wegen lohnt es sich also, bei längeren Namen einen kürzeren zu vergeben, wenn man sich auf eine Bibliothek bezieht. So sorgt “import vizdoom as vzd” dafür, dass später nur noch “vzd” getippt werden muss, wenn man sich auf die Bibliothek “vizdoom” beziehen möchte
- “**from..import..**”: Es gibt große Bibliotheken. Sehr große Bibliotheken. Und gleichzeitig werden nur wenige oder gar nur eine Funktion, quasi nur ein Buch oder Kapitel aus der ganzen Bibliothek benötigt. Das laden riesiger Bibliotheken benötigt aber Rechnerkapazität, somit wird durch das laden einer ganzen Bibliothek, wenn man nur eine oder wenige Funktionen benötigt unnötig Leistung verschenkt. Mit dem oben beschriebenen Befehl wird aber nur das was ich genau spezifiziere geladen. So sorgt also “from __future__ import print_function” dafür, dass nur die print_function geladen wird statt der ganzen Bibliothek. Sollen mehrere Funktionen geladen werden, werden diese einfach durch Kommata getrennt (“from Beispielbibliothek import Funktion1, Funktion2, Funktion3”)¹

Laden des Szenarios: Siehe Kapitel 1.6 und. Es empfiehlt sich, in einer ruhigen Minute einmal durch die Konfigurationsdatei zu stöbern. Vieles dort erklärt sich von selbst.

Probandendaten und Referenzen zu Referenzen von Referenzen:

```
# Enter Subject Data
sub_id = "01"

game = vzd.DoomGame()
```

- Im oberen Teil können Probandendaten eingegeben werden. Aktuell beschränkt sich das (des späteren Titels der Datei wegen um sie auseinanderzuhalten wenn mehr als 1 Person Daten erzeugt) auf die Probanden-Identifikation die wir mit **sub_id** bezeichnen. Aktuell wird hier eine Zahl (durch die “ ” als String also Zeichenfolge gelesen) eingegeben, bevor das Script gestartet wird. Die Information selbst wird insofern nicht aufgezeichnet, als dass damit später lediglich der spezifische Dateiname erzeugt wird, aber in den Daten selbst ist die ID nicht vorhanden. Wollen wir das, und auch noch mehr Daten zu Proband*innen aufzeichnen, müssen wir das Script an mehreren Stellen ändern, später dazu mehr in Kapitel 2.1.

¹Ich bin mir nicht mehr sicher, warum ich die print_function hier importiere, da Python standardmäßig über eine print-Funktion verfügt. Da sich die Zeile aber als Beispiel für die selektive Importierung bestimmter Funktionen eignet, habe ich sie im Code belassen. Gerne darf der ganze Code auch einmal ohne diese importierte Funktion getestet werden. Dazu die Zeile auskommentieren indem man ein “#” davor setzt

- Anhand der Zeile `game = vzd.DoomGame()` lässt sich die Funktionsweise des Codes noch einmal gut erklären: Zuvor haben wir mit “import vizdoom as vzd” schon einmal dafür gesorgt, dass wir nicht immer “vizdoom” eingeben müssen wenn wir auf die Bibliothek zugreifen bzw referieren möchten. Und nun gehen wir einen Schritt weiter: Wir sind in der Bibliothek und möchten uns später auf mehrere Bücher aus einer bestimmten Abteilung beziehen. Die “Abteilung” heißt hier **DoomGame**. Den Punkt zwischen “vzd” und “DoomGame” kann man sich wie das Doppelklicken auf einen Ordner vorstellen. Wir sind also im Ordner vizdoom und klicken doppelt auf den Ordner DoomGame um dort hineinzugelangen. Warum das ganze jetzt als “game” bezeichnen? Ganz einfach: Der Codelänge wegen. Siehe hierzu den übernächsten Abschnitt.

```
# define a number of blocks to write a file after each block so in case
# the game crashes for any reason, not all data is lost
blocks = 2
for b in range(blocks):
```

- Wir legen hier eine Anzahl an Blöcken fest. Grund: Wir wollen die Daten regelmäßig sichern, für den Fall, dass das Spiel zum Beispiel kurz vor Ende des gesamten Experiments abstürzt. Man stelle sich den Frust vor, wenn es nach Versuch 198 von 200 einfriert und in der Folge alle vorherigen Daten nicht gespeichert würden. Und die einfachste Möglichkeit ist, das Ganze in Blöcke zu packen und alles, was innerhalb des Blocks passiert in einen for-Loop zu schreiben. Das heißt die Anzahl der Blöcke bestimmt, wie oft das Spiel geladen wird (in diesem Falle und wenn der Code so funktioniert hat eben 2 mal). Wie viele Versuche es innerhalb des Blocks gibt, bestimmt die Variable “episodes”, wie wir später sehen.

Freischalten von Informationen:

```
# Enables information about all objects present in the current episode/level.
game.set_objects_info_enabled(True)

# Enables information about all sectors (map layout).
game.set_sectors_info_enabled(True)
```

- Hier sehen wir, warum es gut war, unsere Referenz auf die “Abteilung” Doomgame der “Bibliothek” vizdoom mit “game” zu bezeichnen. Sonst müssten wir jetzt nämlich schreiben (wir stellen uns vor, dass auch vizdoom nicht als vzd importiert wurde): `vizdoom.DoomGame.set_objects_info_enabled(True)`. Nicht nur dass das ein Quell vielseitiger Tippfehler ist, da wir beim Programmieren sehr viel spezifisch anweisen müssen, ist jeder gesparte Buchstabe viel wert.
- Funktion der beiden Zeilen: Wir erlauben dem Programm, uns Informationen über die im Spiel vorhandenen und entstehenden Objekte sowie die Karte an sich zu geben. Warum

wir das extra spezifizieren und es nicht automatisch so ist? Es sei nochmal auf den Anfang dieser Sektion hingewiesen, als wir uns mit dem Importieren beschäftigt haben. Je mehr ich das Programm machen lasse, desto mehr Rechnerkapazität wird benötigt die dann vielleicht irgendwann für einen Nachteil in der korrekten Aufzeichnung der Daten nicht mehr zur Verfügung steht (konkret: Weil der Computer so viel auf einmal rechnen muss verpasst er zum Beispiel, dass eine Taste gedrückt wurde beim Aufzeichnen).

Sich eine “weiße Leinwand” schaffen (und direkt wieder darauf malen):

```
# Clear all game variables first to unify the variables for all scenarios
game.clear_available_game_variables()

# Add game variables for Health and Ammo
game.add_available_game_variable(vzd.GameVariable.HEALTH)
game.add_available_game_variable(vzd.GameVariable.AMMO2)

# Add Game Variables for the position
pos_x = game.add_available_game_variable(vzd.GameVariable.POSITION_X)
pos_y = game.add_available_game_variable(vzd.GameVariable.POSITION_Y)
pos_z = game.add_available_game_variable(vzd.GameVariable.POSITION_Z)
angle = game.add_available_game_variable(vzd.GameVariable.ANGLE)
```

- Wir arbeiten hier zunächst mit einem Szenario. Dieses hat von Haus aus (wenn man in die entsprechende .cfg-Datei schaut) bestimmte Variablen. Variablen sind innerhalb dieses Spiels Dinge wie Munition, Gesundheit, Anzahl getöteter Gegner,... . Die Variablen unterscheiden sich teilweise, je nach Szenario. Da wir aber eine einheitliche Datenaufzeichnung anstreben (um später weniger Arbeit zu haben, wenn wir das ganze auf andere Szenarien ausweiten) empfiehlt es sich, die Variablen in diesem Script einheitlich festzulegen. Dass dabei teils auch unveränderliche (und damit für eine weitere Betrachtung unwichtige) Daten erzeugt werden², ist zum Wohle direkt vergleichbarer Datenaufzeichnungen zu vernachlässigen.
- Mit dem “game.clear_...”-Befehl werden nun also alle in der jeweiligen Konfigurationsdatei (.cfg) quasi auf Null gesetzt. Wichtig: Nicht dauerhaft (es wird nichts gelöscht), sondern nur für die Zeit in der wir diese Datei im Rahmen dieses Scripts nutzen. Vielleicht ist also passender zu sagen, die Variablen in der Konfigurationsdatei werden für den Moment ignoriert.
- Mit einer ganzen Reihe von “game.add_...”-Befehlen fügen wir nun unsere gewünschten Variablen ein. Diese sind dann unabhängig vom Szenario eingefügt. Zum Start möchten wir bei allen Szenarien die Gesundheit (HEALTH) und die Menge der Pistolenmunition

²Es ist bspw. nicht nötig, im Szenario “Basic” die Gesundheit des Spielers aufzuzeichnen, da das Monster nicht angreift und sich die Gesundheit des Spielers somit nicht ändert

(AMMO2), sowie die Position und die Orientierung (in Grad) **des Spielers** erfassen. Es gibt eine lange [Liste an möglichen Variablen](#)³, auch diese können zum Experimentieren hinzugefügt und aufgezeichnet werden (für das Prinzip siehe Kapitel 2.1).

Die Optik, Modus und Initialisierung

```
# Set screen size
game.set_screen_resolution(vzd.ScreenResolution.RES_1280X960)

# Enables spectator mode so you can play, but your agent is supposed to watch, not you.
game.set_window_visible(True)
game.set_mode(vzd.Mode.SPECTATOR)
game.init()
```

- Zunächst legen wir die Auflösung und damit die Größe des Spielbildschirms fest. Standardmäßig (also wenn wir die Auflösung (englisch: resolution) nicht festlegen) ist diese über die Konfigurationsdatei mit 320X240 Pixeln festgelegt. Auch hier die Ermutigung, verschiedene Auflösungen/Größen auszuprobieren, aber Achtung: Es muss ein Vielfaches der ursprünglichen Auflösung sein, des Seitenverhältnisses wegen (es gibt sonst einen Fehlercode und das Spiel startet nicht). Im Beispielscript wurde die Auflösung also um den Faktor 4 vergrößert.
- `game.set_window_visible(True)`: Da ViZDoom ursprünglich für computerbasiertes Lernen gedacht war und ist, und der Computer auch visuelle Daten rechnerisch auswertet ist standardmäßig die Sichtbarkeit des Spiels selbst ausgeschaltet. Da wir hier aber aktiv agieren möchten und über keine elektronische Leitung vom Programm zum visuellen System in unserem Hirn haben, müssen wir den Bildschirm anschalten
- Danach legen wir den Modus fest und auch hier zeigt sich, wofür ViZDoom ursprünglich entwickelt wurde: Um aktiv zu spielen müssen wir den Modus auf SPECTATOR also “Beobachter” setzen. Wenn der Computer selbst spielt steht der Modus auf PLAYER also “Spieler”. Der Modus sagt also, was der Computer macht, nicht was wir machen. Und wenn wir spielen, beobachtet der Computer (und greift nicht aktiv ein).
- Zuletzt initialisieren wir das Spiel, starten es also.

Bis hierher haben wir alle Voreinstellungen vorgenommen: Woher bezieht es Daten zur Konfiguration(Karte,Auflösung,Variablen, Spielbare Aktionen,...), wir haben bestimmte Inhalte der Konfiguration nochmals individuell geändert, dem Programm erlaubt, uns Daten für Positionen und Objekte freizugeben, den (virtuellen) Bildschirm angeschalten, festgelegt was der Computer macht (beobachten) und das Spiel gestartet.

³Auf der Website bis zu einer Auflistung die mit “KILLCOUNT” beginnt nach unten scrollen

Ab hier geht es nun darum, was **innerhalb** des Spiels passiert: In welchem Rahmen und wie Daten erzeugt und abgerufen/ausgegeben werden, wie lange ein Spiel geht und wie die Daten letztendlich abgespeichert werden.

Funktionen: Zunächst ein kurzer Exkurs zu Funktionen im allgemeinen. Funktionen können sehr hilfreich in der Vermeidung von unübersichtlich langem Code sein. Nicht nur, dass sie eingeklappt werden können, sodass man sie nicht immer in Gänze entlang gehen muss bei Betrachtung des Scripts, sondern auch, wenn es an die in den Funktionen definierten Aktionen in der Organisation geht. Wenn ich eine Aktion^[c] immer wieder ausführen möchte, so empfiehlt es sich, diese einmal zu definieren und später nur noch zu spezifizieren womit es denn genau arbeiten soll. Etwas klarer wird es vielleicht wenn wir das nochmal an den beiden nachfolgend im Script definierten Funktionen betrachten ^[c]: Hier im Sinne von “du bekommst einen Input und machst etwas damit”

Bewegungen in geeignetes Format übersetzen:

```
# Define a translation function for the last action
def translate_action(last_action):

    last_action_trnsl = "NA"
    #scenario basic

    if last_action == [1.0,0.0,0.0]:
        last_action_trnsl = "MOVE_LEFT"
    elif last_action == [0.0,1.0,0.0]:
        last_action_trnsl = "MOVE_RIGHT"
    elif last_action == [0.0,0.0,1.0]:
        last_action_trnsl = "ATTACK"

    return last_action_trnsl
```

- Was ich immer zuerst mache ist, eine Funktionsdefinition starten (mit “def”) dann der Funktion einen Namen geben (hier “translate_action”) und dahinter in Klammern den Input. Ich brauche hier noch keinen konkreten Verweis auf etwas bestimmtes, d.h. ich vergebe hier einen Namen als Platzhalter. Da ich nacher als Input die letzte vom Computer beobachtete Bewegung (=Action) möchte, nenne ich den Platzhalter “last_action”. Später, also wenn ich die Funktion konkret benutze, spezifiziere ich dann, was ich mit “last_action” meine.
- Wie oberhalb des Codes beschrieben, möchte ich etwas “übersetzen”. Ich weiß, dass die Bewegungen des Szenarios über drei Vektoren definiert werden. Und diese möchte ich jeweils in Worte übersetzen, um später in der Datenaufzeichnung direkt lesen zu können, welche Bewegung ausgeführt wurde.

- Zunächst definiere ich die Variable die ich nachher auswerfen lassen will (`last_action_trnsl`) mit einem leeren Wert. Ich nehme hier den String “NA”. Die Variable vorerst mit einem Wert zu versehen ist notwendig.
- Und dann sage ich der Funktion wie es mit dem Input umgehen soll: durch `if/elif`-Funktionen sage ich jetzt “Wenn ‘last_action’ dem Vektor [x,y,z] entspricht, dann ersetze den Wert”NA” von ‘last_action_trnsl’ mit “ABC”.” Wenn der erste Fall nicht zutrifft, geht es über “elif” zum nächsten usw..
- Schlussendlich soll es mir die Variable ‘last_action_trnsl’ zurückgeben (englisch “to return”).
- Kurz und knapp: Die Funktion nimmt als Input ‘last_action’, gleicht diesen Input mit den 3 angegebenen ab und ändert den Wert der Variable ‘last_action_trnsl’ von “NA” zu dem entsprechenden String, und gibt die Variable mit dem neuen Wert zurück. Bewege ich mich oder schieße, ist der Input gleich einem der drei Vektoren, wenn ich nichts tue, bleibt der Wert von ‘last_action_trnsl’ bei “NA”

Objektdaten erzeugen

```
#function to return objects info
def find_object_data(object_list,object_id,object_name):
    for o in object_list:
        if o.id == object_id and o.name != object_name:
            return o
    return None
```

- Hier hat die Funktion den Namen “find_object_data” und drei Variablen als Input. Ich möchte später mit einer Liste, einer spezifischen ID und einem bestimmten Objektnamen arbeiten, also nenne ich die Platzhalter für die Variablen entsprechend.
- `for ... in ...` : Ich möchte jetzt dass es für jedes Objekt in der Liste etwas macht. ‘object_list’ ist hier genauso zu verwenden, aber das ‘o’ könnte theoretisch auch ‘i’ oder ‘e’ oder ‘object’ heißen. Da es mir um ein Objekt geht und ich Platz sparen möchte habe ich ‘o’ gewählt. Allgemein sagt dieser Teil aber “Für jedes Element in diesem oder jenem”.
- Ich brauche später nur noch Daten der anderen Objekte neben dem Spieler (der Spieler ist für das Spiel ein Objekt). Darum sage ich “Wenn die Objekt_ID die du aus dem Spiel ziehst gleich der ‘object_id’ ist und wenn gleichzeitig der Objektname ungleich einem bestimmten ‘object_name’ ist, dann:”
- “gebe mir das Objekt (seine Daten) zurück” -“ansonsten gebe nichts zurück”

Spiellänge:

```
# Specify how many Episodes
episodes = 10
```

- Wir sagen hier, wie viele Episoden gespielt werden, bis das Spiel geschlossen wird. Eine Episode entspricht einem Spieldurchlauf, der Spieldurchlauf wird wiederum beendet, indem eines von mehreren oder ein festgelegtes Ziel erreicht wird. Bspw. endet im Szenario “Basic” eine Episode, sobald das Monster getötet (erschossen) wurde, oder nach 151 Schleifendurchläufen (später dazu mehr). Im Beispiel ist die Anzahl der Episoden auf 10 festgesetzt. Wir erinnern uns: Es gibt zwei Blöcke, also $2 \cdot 10$ Episoden.

Die Datei anlegen:

```
# header for the csv-file
columns = ["Episode",
           "State",
           "Tic",
           "Health",
           "Ammo",
           "x_pos",
           "y_pos",
           "z_pos",
           "angle/orientation",
           .
           .
           .
          ]
```

- Wir erstellen eine Liste mit ‘[]’ um darin die Spaltennamen für **alles innerhalb der Episoden** zu speichern. Hier muss also schon die Überlegung stattfinden, welche Daten man in welcher Reihenfolge aufzeichnen möchte. Die Liste nennen wir “columns” (Spalten)

Die Datei schreiben

```
# Open a CSV file to store episode-data of each block

block_filename = sub_id + "_game_data_block_" + str(b+1) + ".csv"
with open(block_filename, 'w', newline='') as starting_file:
    csv_writer = csv.writer(starting_file)
    csv_writer.writerow(columns)
```

- Theoretisch könnte sowohl die Liste mit den Spaltennamen als auch das Schreiben der Datei erst hinterher geschehen, es ist aber arbeitstechnisch sinnvoller, da durch die Liste direkt vor der Erstellung der Datenaufzeichnung einen Überblick über die zu erstellenden arrays gibt (gleich mehr dazu).

- Was diese Zeilen Code nun machen: Zunächst erstellen wir einen Dateinamen. Die Variable dafür nennen wir passenderweise ‘block_filename’. Und dann sagen wir aus was der Dateiname für jeden Block bestehen soll: aus der Variable ‘sub_id’, welcher wir bereits zu Beginn einen Wert geben (dieser dient dazu Daten verschiedener Personen auseinander zu halten, wenn man mehrere Personen misst), zu dem Wert dieser Variable einen String (den wir deshalb in “ ” packen), außerdem die Zahl des Blocks (was eigentlich eine Zahl ist, ich möchte es hier aber als String, darum schreibe ich str() davor. Außerdem ist zu bedenken, dass Python mit dem Zählen bei 0 beginnt, weshalb ich immer eine 1 dazu addieren muss). Letztendlich darf die Dateiendung “.csv”⁴ nicht fehlen.
- dann “öffnen” (with open) wir die Datei und sagen zum einen dass sie schreibbar (writeable = ‘w’) ist und mit welchem Namen wir sie öffnen (as starting_file). Anschließend sagen wir mit den zwei weiteren Zeilen in Kürze gefasst, dass es unsere Spaltennamen in einere Reihe in die Datei schreiben soll. Wenn wir uns diese Datei also jetzt ansehen würden, hätten wir eine Zeile mit 25 Zellen die die jeweiligen Begriffe beinhalten.

Weitere Spalten, eine weitere Datei:

```
#start with columns-header for the Episode_End_Data
columns2 = ["Episode",
            "Total_Reward",
            "Time",
            "FPS"]

# Open another csv-file
endrow_filename = sub_id + "_endrows_block_" + str(b+1) + ".csv"
with open(endrow_filename, 'w', newline='') as end_file:
    csv_writer = csv.writer(end_file)
    csv_writer.writerow(columns2)
```

- Hierzu möchte ich gar nicht zu viele Worte verlieren, da es sich hier genau wie oben verhält. Was hiermit erreicht wird: Neben den Dateien für alles was innerhalb der Episoden eines Blocks passiert, lassen wir uns jetzt zu Ende jeder Episode die wichtigsten Daten zusammenfassen: Episodenzahl, Gesamtpunkte, Gesamtzeit, FPS, also wie viele Bilder pro Sekunde

Doppelung von Informationen und Chaos in den Daten vermeiden:

⁴csv steht für “comma separated value”, also Daten die durch Kommata getrennt sind. Das wird vor allem wichtig, wenn wir die Daten später einlesen und als Tabelle darstellen wollen. Dann müssen wir nämlich sagen, woran das Programm erkennen kann, dass eine Zelle zu Ende ist.

```
#Use a set to keep track of uniqueobject IDs
unique_object_ids = set()
```

- In Kürze erklärt: Diese Zeile erstellt mir eine zunächst noch leere Menge. Später füllen wir diese mit konkreten IDs von Objekten. Da aber in der Abfrage manche Objekte dauerhaft also damit immer wieder vorkommen, würde die Menge immer größer werden und die ganze Datenaufzeichnung ins Chaos stürzen. Diese Menge ist jedoch anders: Sie lässt keine Doppelungen zu. Wenn ich also später versuche einen Wert hinzuzufügen, der schon in der Menge an Werten ist, wird dieser Wert einfach nicht hinzugefügt.

Arrays erstellen für die spätere Datenaufzeichnung:

```
#create single column arrays for each variable to later concatenate into a dataframe
Episode_arr = np.full((episodes*151,1), np.nan, dtype = np.int32)
State_arr = np.full((episodes*151,1),np.nan, dtype = np.int32)
Tic_arr = np.full((episodes*151,1), np.nan, dtype = np.int32)
Health_arr = np.full((episodes*151,1),np.nan, dtype = np.int32)
.
.
.
```

- Arrays sind so etwas wie Arbeitsspeichergefäße. Wir benutzen Arrays, weil wir so die höchstmögliche Geschwindigkeit der Datenaufzeichnung des Spieldurchlaufes haben. Warum? Arrays werden vor der Datenaufzeichnung erstellt und in ihrer Größe definiert. Ich sage also vorab schon genau, wie viel Arbeitsspeicher der Computer bereit halten soll. Das erhöht die Geschwindigkeit weil der Computer eben nicht ständig prüfen muss, ob er den Speicher vergrößern muss (und wenn er merkt er muss es, wird das Erweitern selbst ja auch nochmal eine zeitraubende Angelegenheit. Da wir hier hier im Bereich von 10-30 ms arbeiten, müssen wir alles was zeit kostet und sei es noch so wenig, verhindern).
- Mit der Bibliothek numpy gestaltet sich das “Bauen” des zunächst leeren Arrays wie folgt: `name_des_Arrays = np.full((Zeilen,Spalten), np.nan5, Datentyp6)`
- Warum hier jetzt 25 einzelne Arrays mit jeweils einer Spalte erstellen? Unter anderem, weil das erstellen von Arrays mit gemischten Datentypen (sogenannten “structured arrays”) schnell komplex wird.
- Hier zeigt sich jetzt auch, warum es sinnvoll ist, die Spaltennamen bereits vorab festzulegen: Ich kann mich nun an dieser Liste entlang bewegen beim Erstellen der Arrays um so nichts zu vergessen.

⁵np.nan steht hier für “numpy not a number”. Es ist einfach nur ein leerer Platzhalter

⁶Im Script sieht man mehrere Datentypen: np.int32 = integer 32 bit (ganze Zahl), np.float = Kommazahl 64 bit, und der “Sonderfall” für Strings, also Buchstaben, np.dtype(‘Uxy’), wobei xy für eine Zahl steht, die definiert, wie viele Zeichen **maximal** im String sind.

- Das gleiche machen wir dann noch für die Datei die unsere zusammengefassten Daten enthält. (Siehe das Script)

Start-Index

```
#starting the index
current_index = 0
```

- Hier wird der Start-Index definiert als `current_index` (momentaner Index). Dies dient vor allem dem geordneten befüllen der Arrays wie wir gleich sehen. Und damit er sich innerhalb der Episode erhöht aber nicht zwischen den Episoden wird er *vor* den Episodenloop gesetzt und damit nach jeder Episode wieder auf Null gesetzt.

Bis hierhin ist jetzt schon viel passiert, ohne das überhaupt “etwas” passiert ist. Wir haben dem Programm bisher gesagt, dass und wie groß wir das Spiel sehen wollen, welche Variablen es gibt, welche Steuereinstellung und welches Spielszenario wir verwenden möchten, dass es bitte zwei Dateien anlegen soll und ihm gesagt, eine bestimmte Größe an Arbeitsspeicher bereitzustellen um diesen mit Daten zu füllen und den Computer so um die zeitraubende Aufgabe zu entlasten, zu prüfen ob mehr Speicher notwendig ist und diesen dann erzeugen. Aber jetzt geht es darum, was innerhalb eines Blocks und dann ziemlich schnell auch schon darum, was innerhalb einer Episode dieses Blocks passiert.

Episoden-Loop:

```
# Loop through episodes
for i in range(episodes):
```

- Das hier ist so gesehen schon der erste Loop im Loop (wir erinnern uns an den Block-Loop). Was sagt uns diese Zeile nun? Sie sagt nichts anders als “Für jedes Element `i` in der Zahlenreihe⁷ (`range`) der episodes tust du folgendes:”

```
print("Episode #" + str(i + 1))
    game.new_episode()
    cumulative_reward = 0 # Initialize cumulative reward
    start_time = time.time() # Record the start time
```

- Die erste Zeile ist eher zur Information. Es sagt dem Programm was es zu Beginn in die Konsole schreiben soll. Das kann aber z.B. hilfreich sein, wenn aus irgendwelchen Gründen das Fenster mit dem Spiel nicht zu sehen ist. Dann kann man an dieser Zeile in der Konsole sehen, dass das Spiel gestartet hat

⁷Eine wörtliche Übersetzung ist hier schwierig. Wir haben mit `episodes` eine Zahl definiert. Ob es sich hier nun bei einzelnen Werten `i` um Zahlen handelt ist egal. Das Programm weiß: `episodes` gibt mir an wie viele Episoden bzw Elemente es gibt.

- “game.new_episode()” startet die Funktion des ViZDoom Codes der in seiner Tiefe gar nicht näher behandelt wird, aber eben eine neue Episode startet
- in den zwei weiteren Zeilen setzen wir die kumulative⁸ Punktrechnung und sagen mit “start_time = time.time()” dass die Zeit die der Computer in diesem Moment misst unsere Startzeit ist.

Der While-Loop:

```
while not game.is_episode_finished():
```

- Nicht verwirren lassen, aber das ist tatsächlich der (While-)Loop im (For-)Loop im (For-)Loop.
- Hier haben wir jetzt eine neue Art des Loops wie darüber schon zu lesen ist. Ein While-Loop ist eine Schleife, die solange ausgeführt wird bis etwas bestimmtes passiert.
- Die Zeile sagt also “Solange ‘game.is_episode_finished()’ noch nicht passiert ist tue folgendes:”. Dabei ist ‘game.is_episode_finished()’ eine Abfrage des Programms, die in unserem Beispiel dann “Ja” bzw. “TRUE” zurückmeldet wenn das Monster besiegt ist, weil dass die Bedingung dafür ist. (Aber auch das bewegt sich in den tiefen des Codes)

Ein paar Worte zum folgenden Inhalt des Loops vorab: Was wir jetzt beschreiben sieht nach sehr viel aus aber beschreibt nur was in wirklich kurzer Zeit passiert (wir sprechen hier von ca 30 ms die der Computer braucht um die Schleife einmal zu durchlaufen). Es geht jetzt darum welche Daten abgefragt und wo sie zwischengespeichert werden (Arbeitsspeicher also die Arrays wie wir im Folgenden sehen)

1.8 To follow

- Was passiert mit den Daten und wo werden sie gespeichert und wie schaut man sie an?
- Was genau wird da aufgezeichnet?
- More Data: Script anpassen um bspw. sub_id in Tabelle, sowie Geschlecht, Alter,....
- More Data: Auch mehr Variablen einfügen und Datenaufzeichnung anpassen

⁸kumulativ so etwas wie “hinzufügen”, wir addieren fortlaufend die neuen Werte der Punktzahl zu diesem Wert

2 Die Daten: Format, Aussage, Umgang

2.1 Noch mehr Daten aufzeichnen

Referenzen

Article. o. J.

craigloewen-msft. 2023. „Installieren von WSL“. <https://learn.microsoft.com/de-de/windows/wsl/install>. „Developing in the Windows Subsystem for Linux with Visual Studio Code“. o. J. <https://code.visualstudio.com/docs/remote/wsl>. Zugegriffen 6. Oktober 2023.

Gite, Vivek. 2018. „How Do I Update Ubuntu Using Terminal Command Line“. *nixCraft*. <https://www.cyberciti.biz/faq/upgrade-update-ubuntu-using-terminal/>.

Hameed, Sharqa. 2023. „How to Install Pip on Ubuntu 22.04“. *Linux Genie*. <https://linuxgenie.net/how-to-install-pip-on-ubuntu-22-04/>.

Kempka, Michal, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, und Wojciech Jaskowski. 2016. „ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning“. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8. Santorini, Greece: IEEE. <https://doi.org/10.1109/CIG.2016.7860433>.