Wydział Elektroniki i Technik Informacyjnych
Politechnika Warszawska

Modelowanie i Symulacja Komputerowa: Kolonizacja Marsa – łaziki monitorujące warunki glebowe na polach uprawnych

Projekt

Piotr Chęciński, Jakub Kalankiewicz, Kacper Marchlewicz, Mateusz Sobaś, Wojciech Styczeń

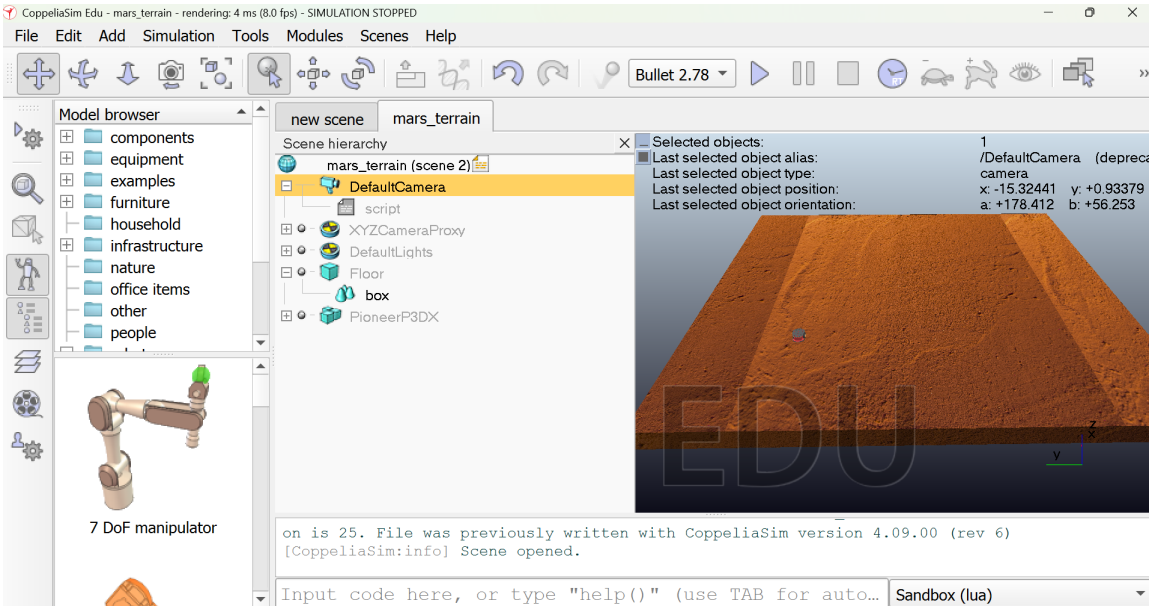
Warszawa, 2025

Spis treści

1. Część 1 - instalacja środowiska do symulacji, projekt systemu, modele komponentów, algorytmy

1.1. Wybrane środowisko

Symulacja zostanie zrealizowana w środowisku CoppeliaSim (wcześniej znanym jako V-REP) zaprojektowanym przez przedsiębiorstwo Toshiba, a obecnie utrzymywanym i rozwijanym przez szwajcarskie przedsiębiorstwo Coppelia Robotics AG. CoppeliaSim to zaawansowane środowisko służące do modelowania i symulowania złożonych systemów wykorzystujących roboty. CoppeliaSim udostępnia bogaty interfejs API, który pozwala wykonywać wiele operacji w środowisku CoppeliaSim z poziomu kodu w wybranym języku programowania, np. Python, C++, MatLab, Rust, Lua. W projekcie wykorzystany zostanie język programowania Python. W bezpośrednim wykorzystaniu API CoppeliaSim w Python konieczne jest użycie biblioteki ZeroMQ. Znaczącą zaletą tego rozwiązania jest nieskomplikowana konfiguracja. Na rysunku 1 przedstawiono zrzut ekranu z uruchomionym środowiskiem CoppeliaSim.



Rysunek 1: Zrzut ekranu z uruchomionym środowiskiem CoppeliaSim (*źródło:* opracowanie własne).

Na listingu 1. przedstawiono kod w Pythonie, który pozwala na uruchomienie symulacji na obecnie otwartej scenie w CoppeliaSim. Zaletą wykorzystania biblioteki ZeroMQ jest to, że skrypt Python automatycznie wykrywa uruchomione środowisko CoppeliaSim na komputerze. Użytkownik nie musi ręcznie konfigurować połączenia Python-CoppeliaSim.

```
1 from coppeliasim_zmqremoteapi_client import RemoteAPIClient
2
```

```
client = RemoteAPIClient()
sim = client.require('sim')

sim.startSimulation()
```

Listing 1: Kod w Pythonie uruchamiający symulację w CoppeliaSim (źródło: `https://manual.coppeliarobotics.com/en/remoteApiOverview.htm`, data dostępu: 15.03.2025)

1.2. Projekt systemu

Symulacja: Co z góry określony cykl czasu (np. 30 sekund) będzie następowała aktualizacja parametrów 'zmiennych', takich jak stan baterii (np. łazik co 1 minutę traci 1% baterii), czy też wilgotności pola. Co cykl będą więc wywoływane odpowiednie funkcje. Dodatkowo zaimplementowany zostanie mechanizm dnia i nocy, co przełoży się na inne warunki pracy łazików i zmiany parametrów pól uprawnych.

Centrala: Zadaniem centrali jest przechowywanie informacji o polach uprawnych oraz wysyłanie łazików do wykonania konkretnego zadania np. nawodnienia poprzez wystawienie odpowiedniego zadania. Centrala posiada listę pól, w których każdy element to współrzędne środka oraz parametry gleby. Jeśli któryś z parametrów jest za niski, to zostanie wystawione dla łazików odpowiednie zadanie. Co pewien cykl czasu (np. co 8 godzin) centrala wystawi zadania sprawdzenia stanu pól.

Łaziki: Łazik kołowy będzie wyposażony w kamerę (RGB/RGBD). Łaziki poruszają się wyznaczając scieżki za pomocą wybranego algorytmu. Posiadają także funkcję przejścia w tryb ładowania, (wtedy łazik rozkłada swoje panele słoneczne) co czynią gdy ich stan baterii osiągnie np. 15%. Gdy tak się stanie, łazik przerywa swoje obecne zadanie, informując o tym centralę (zadanie zostaje zwolnione, aby inny łazik mógł je wtedy wykonać). Gdy łazik rozpocznie ładowanie, będzie się ładował do pełnego stanu baterii. Łaziki potrafią rozpoznawać znaczniki ArUco. Łazik gdy nie ma zadania to przyjmuje wolne zadanie wystawione przez centralę. Gdy łazik przyjmie zadanie to przełącza się w tryb pracy. Gdy nie będzie wolnego zadania to oczekuje w trybie uśpienia. Tryb uśpienia charakteryzuje się niższym zużyciem energii od trybu pracy (np. o połowę).

Pola: Pola są generowane losowo, na środku pola znajduje się wskaźnik ArUco, co umożliwi łazikom (w scenariuszu potrzeby mapowania) odnalezienie środków pól. Wymiary przyjmujemy z góry określone. Pola to klasy zawierające informacje o swoich współrzędnych, a także inne parametry jak wilgotność, czy pH które mogą się zmieniać w czasie.

1.3. Modele komponentów

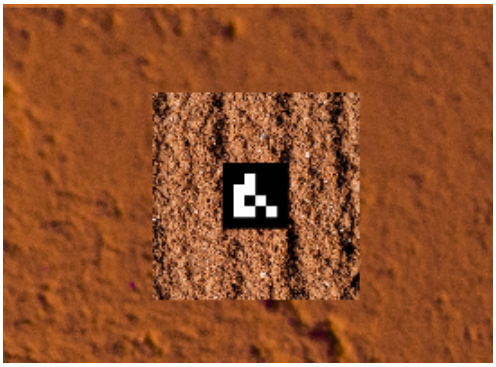
W symulacji występują trzy główne komponenty: pola uprawne, łaziki i centrala. Każdy z nich został poniżej szczegółowo opisany.

1.3.1. Model pola uprawnego

Opis funkcjonalny: Model pola uprawnego reprezentuje pola uprawne, które są monitorowane przez łaziki. Każde pole jest opisywane przez następujące parametry:

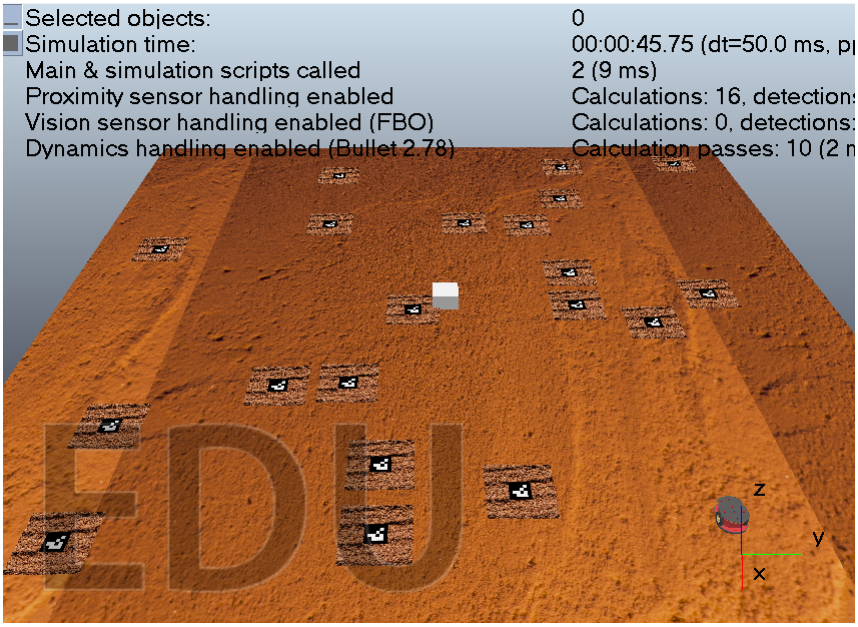
— położenie,
— wilgotność (humidity),
— poziom pH,
— wskaźnik mikrobiomu (microbiome),
— temperatruę,
— wskaźnik składu mineralnego (mineral composition).

W środowisku symulacyjnym każde pole jest o wymiarach 1×1 metr. Dodatkowo na środku każdego pola wygenerowany jest marker ArUco, który będzie wykorzystywany przez łaziki do lokalizacji i nawigacji. Na rysunku drugim przedstawiono przykład pola uprawnego.



Rysunek 2: Zrzut ekranu z uruchomionym środowiskiem CoppeliaSim przedstawiającym przykładowe pole uprawne z markerem ArUco(*źródło:* opracowanie własne).

Struktura i implementacja: Logika pól uprawnych została zaimplementowana w języku Python z wykorzystaniem biblioteki `coppeliasim_zmqremoteapi_client`.Na poziomie kodu określa się liczbę pól, a wszystkie pola generowane są automatycznie (uwzględniona została weryfikacja, aby pola na siebie nie nachodziły). Obecnie wszystkie parametry pól uprawnych generowane są w sposób losowy. Na rysunku 3 przedstawiono przykład symulacji z wygenerowanymi 20 polami uprawnymi. Na obecnym etapie projektu markery ArUco nie są jeszcze unikalne, jednak docelowo do każdego pola będzie przypisany unikalny kod.

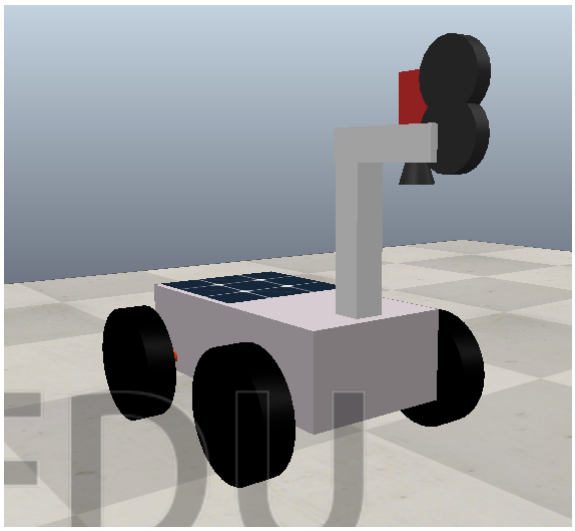


Rysunek 3: Zrzut ekranu z uruchomionym środowiskiem CoppeliaSim przedstawiającym przykładowych 20 pól uprawny z kodem ArUco (*źródło:* opracowanie własne).

Wnioski: Model pól uprawnych został zaprojektowany jako osobny komponent o określonych parametrach, teksturze i kodzie ArUco. W dalszej fazie projektu implementacja pól uprawnych w symulacji zostanie dopracowana, tak aby każde pole posiadało unikalny marker ArUco.

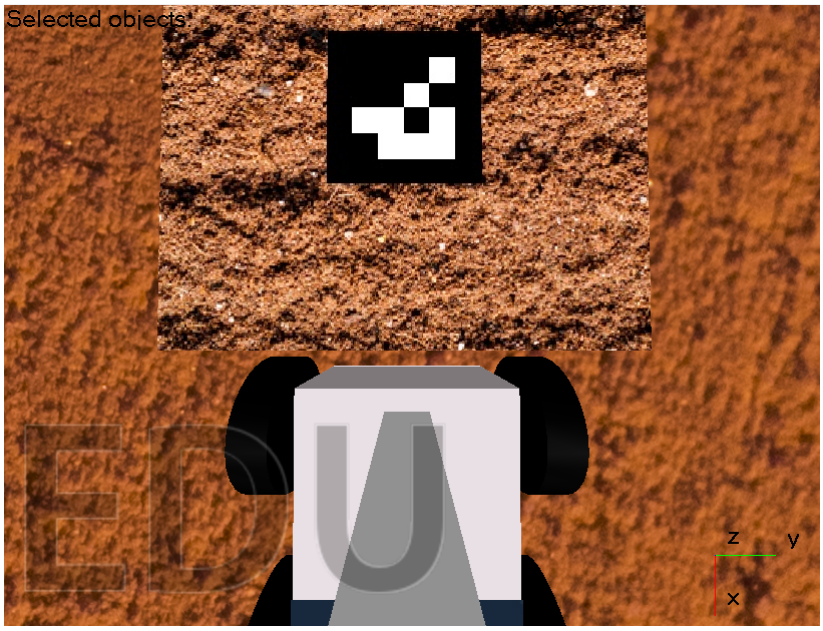
1.3.2. Model łazika

Opis funkcjonalny: Zadaniem łazika w rozpatrywanej symulacji jest monitorowanie stanu pól uprawnych (określonych parametrów) na powierzchni Marsa. W tym celu łazik musi mieć możliwość poruszania się po terenie. Dodatkowo łazik wyposażony jest w baterię, która po wyczerpaniu jest ładowana za pomocą paneli fotowoltaicznych. Na rysunku 4 przedstawiono wstępny model łazika przygotowany w środowisku symulacyjnym.



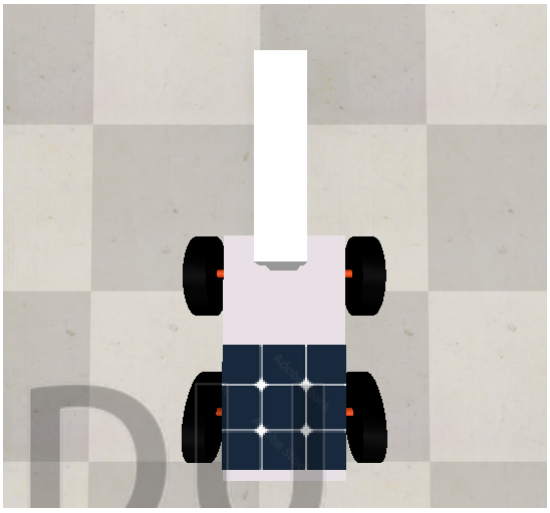
Rysunek 4: Zrzut ekranu z uruchomionym środowiskiem CoppeliaSim przedstawiającym wstępny model łazika (*źródło:* opracowanie własne).

Istotnym elementem łazika jest kamera umieszczona na ramieniu, która pozwoli na wyświetlanie obrazu w celu zczytywania kodów ArUco z pól uprawnych. Na rysunku 5 przedstawiono widok z kamery w pobliżu przykładowego pola uprawnego.



Rysunek 5: Widok z kamery zamontowanej na łaziku przedstawiającej przykładowe pole uprawne (*źródło:* opracowanie własne).

Dodatkowo na obecnym etapie panel fotowoltaiczny został umieszczony na górnej części obudowy łazika, co zobrazowano na rysunku 6.



Rysunek 6: Widok na panel solarny umieszczony na łaziku (*źródło:* opracowanie własne).

Wnioski: W ramach obecnego etapu przygotowano model łazika w środowisku symulacyjnym. W ramach dalszych prac zostanie przygotowana klasa łazika w Python, która będzie przechowywać parametry konkretnego łazika np. stan baterii. W symulacji zakłada się występowanie więcej niż jednego łazika jednocześnie. Łaziki będą komunikować się z centralą w celu otrzymywania zadań i przekazywania informacji.

1.3.3. Model centrali

Opis funkcjonalny: Model centrali stanowi centralny moduł sterująco-monitorujący w systemie symulacji kolonizacji Marsa. Jego podstawowym zadaniem jest monitorowanie parametrów glebowych na polach uprawnych i generowanie odpowiednich zadań dla łazików w przypadku wykrycia nieprawidłowości. Centrala działa w trybie cyklicznym – w każdym cyklu sprawdza stan wszystkich pól uprawnych i podejmuje decyzje dotyczące konieczności interwencji. W środowisku symulacyjnym CoppeliaSim centrala jest reprezentowana jako prosty obiekt o kształcie sześcianu, umieszczony w centralnym punkcie obszaru uprawnego. Sześcian ten został dynamicznie utworzony w czasie uruchomienia symulacji za pomocą API z poziomu skryptu w języku Python.
Struktura i implementacja: Logika centrali została zaimplementowana w języku Python z wykorzystaniem biblioteki `coppeliasim_zmqremoteapi_client`. Model centrali opiera się na klasie `Centrala`, której schemat działania przedstawiono poniżej:

— **Połączenie z symulacją:** Centrala nawiązuje połączenie z uruchomioną instancją CoppeliaSim za pomocą ZMQ Remote API.
— **Utworzenie obiektu w symulacji:** W momencie uruchomienia centrali tworzony jest obiekt typu `primitiveshape_cuboid`, ustawiany jako statyczny element sceny, z nadaną nazwą `CentralaCube`.
— **Wczytywanie informacji o polach:** Centrala wyszukuje w scenie wszystkie pola uprawne na podstawie konwencji nazw (`Field_X`). Dla każdego pola pobiera współrzędne oraz dane glebowe zapisane w atrybucie `CustomDataBlock`.
— **Walidacja danych:** Przed przetworzeniem danych następuje walidacja obecności wszystkich wymaganych parametrów (`humidity`, `pH`, `microbiome`, `temperature`, `minerals`).
— **Monitoring pól:** Centrala cyklicznie (co 5 sekund) sprawdza stan pól. W przypadku wykrycia zbyt niskiej wilgotności (<50%) lub zbyt niskiego pH (<6.0) generowany jest komunikat o potrzebie interwencji.

Kod źródłowy modelu centrali:
Poniżej przedstawiono fragment kluczowej implementacji klasy `Centrala`:

```
class Centrala:
    def __init__(self):
        self.client = RemoteAPIClient()
        self.sim = self.client.require('sim')
        self.fields = []
        self.running = True
        self.centrala_handle = None
        self._create_centrala_cube()
        self._load_fields_from_scene()

    def _create_centrala_cube(self):
        size = [0.5, 0.5, 0.5]
        position = [0, 0, size[2] / 2]
        options = 0
        self.centrala_handle = self.sim.createPrimitiveShape(
            self.sim.primitiveshape_cuboid, size, options
        )
        self.sim.setObjectPosition(self.centrala_handle, -1, position)
        self.sim.setObjectInt32Param(
        self.centrala_handle, self.sim.shapeintparam_static, 1)
        self.sim.setObjectName(self.centrala_handle, "CentralaCube")

    def _load_fields_from_scene(self):
        index = 0
        while True:
            name = f"Field_{index}"
            try:
                handle = self.sim.getObjectHandle(name)
```

```
                pos = self.sim.getObjectPosition(handle, -1)
                data_raw = self.sim.readCustomDataBlock(handle, "SoilData")

                if not data_raw:
                    raise ValueError(f"No SoilData found for {name}")

                data = json.loads(data_raw)
                required_keys = ["humidity", "pH", "microbiome", "temperature",
                "minerals"]
                for key in required_keys:
                    if key not in data:
                        raise KeyError(f"Missing '{key}' in SoilData
                        for {name}")

                area = Area(
                    pos[0], pos[1], pos[2],
                    data["humidity"],
                    data["pH"],
                    data["microbiome"],
                    data["temperature"],
                    data["minerals"]
                )
                self.fields.append(area)
                index += 1
            except Exception:
                break

    def monitor_fields(self):
        for field in self.fields:
            if field.humidity < 50:
                print(f"Field at ({field.x}, {field.y}) needs watering!")
            if field.pH < 6.0:
                print(f"Field at ({field.x}, {field.y}) pH too low!")

    def periodic_check(self):
        while self.running:
            self.monitor_fields()
            time.sleep(5)

    def stop(self):
        self.running = False
```

Listing 2: Klasa Centrala – model centrali monitorującej (*źródło:* opracowanie własne).

Wnioski: Model centrali został zaprojektowany jako osobny moduł monitorujący stan pól uprawnych w czasie rzeczywistym. Dzięki implementacji walidacji danych oraz komunikatów o stanie pól, centrala umożliwia podejmowanie decyzji o konieczności interwencji przez łaziki. Dodatkowo dynamiczne pobieranie danych z atrybutów obiektów w symulacji umożliwia elastyczne dostosowanie się do zmieniającej się sceny.

1.4. Algorytmy

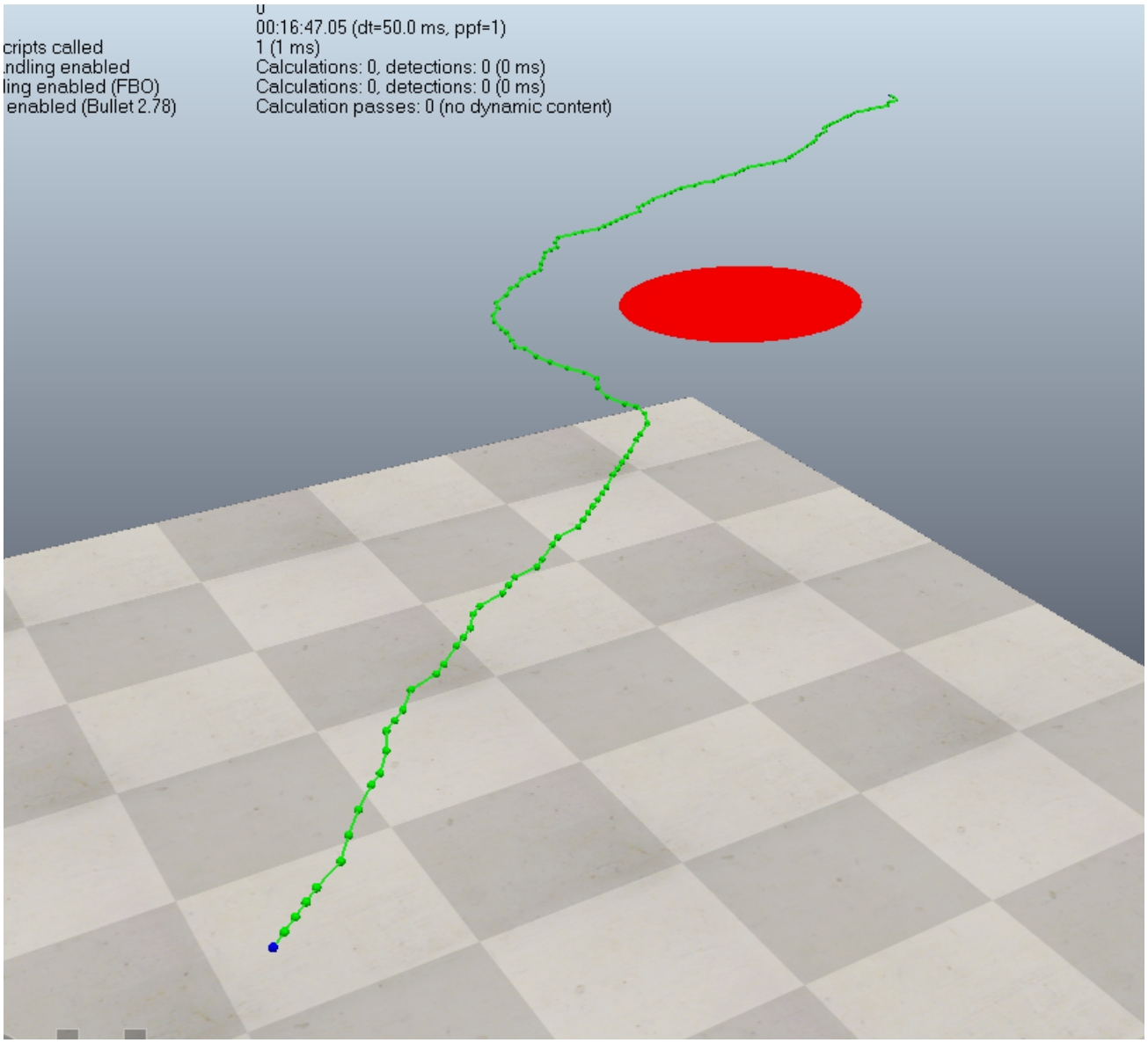
RRT*: rozszerzenie podstawowego RRT (ang. rapidly exploring random tree), algorytm ten jest probabilistyczną metodą planowania ścieżek. Według źródeł jest skuteczny w środowiskach o wysokiej wymiarowości lub z dynamicznymi przeszkodami, co czyni go odpowiednim dla naszej symulacji, z uwagi na potrzebę wyznaczania tras między polami uprawnymi oraz dynamicznej aktualizacji ścieżek w przypadku awarii łazików. Algorytm rozpoczyna od węzła startowego, a następnie w każdej iteracji losowo generuje punkt w przestrzeni, do którego szuka najbliższego węzła, już obecnego w drzewie. Po wyznaczeniu połączenia sprawdzane są kryteria, takie jak możliwość nawiązania połączenia bez kolizji z przeszkodami oraz ocena kosztu przejścia. Kluczową cechą RRT* jest mechanizm „rewiring", czyli ponownego łączenia węzłów. Jeśli nowo dodany punkt umożliwia znalezienie tańszej ścieżki do istniejącego węzła, drzewo jest modyfikowane poprzez aktualizację połączeń, co pozwala na stopniowe udoskonalanie całej struktury. W wyniku tego procesu, przy wystarczającej liczbie iteracji, znaleziony wynik zbliża się do ścieżki optymalnej, co jest szczególnie ważne w środowiskach, gdzie przestrzeń jest rozległa i złożona. Pseudokod przedstawia rysunek 7.

Algorithm 2.
T = (V, E) ← RRT*(z_{ini})

1 $T \leftarrow$ InitializeTree();
2 $T \leftarrow$ InsertNode(Ø, z_{init}, T);
3 ***for*** i=0 to i=N ***do***
4 $z_{rand} \leftarrow$ Sample(i);
5 $z_{nearest} \leftarrow$ Nearest(T, z_{rand});
6 (z_{new}, U_{new}) ← Steer ($z_{nearest}$, z_{rand});
7 ***if*** Obstaclefree(z_{new}) ***then***
8 $z_{near} \leftarrow$ Near(T, z_{new}, |V|);
9 $z_{min} \leftarrow$ Chooseparent (z_{near}, $z_{nearest}$, z_{new});
10 $T \leftarrow$ InsertNode(z_{min}, z_{new}, T);
11 $T \leftarrow$ Rewire (T, z_{near}, z_{min}, z_{new});
12 ***return*** T

Rysunek 7: Algorytm RRT* (źródło: *Iram Noreen, Amna Khan, Zulfiqar Habib (2016): A Comparison of RRT, RRT* and RRT*-Smart Path Planning Algorithms*)

Zaimplementowano w Python algorytm RRT* zgodnie z powyższym pseudokodem. Algorytm przyjmuje punkt początkowy, punkt docelowy oraz listę współrzędnych przeszkód i ich promieni. Wyznaczana ścieżka wygląda poprawnie, algorytm będzie jeszcze wymagał dostrojenia parametrów ogólnych (takich jak liczba iteracji, wielkość kroku, promienie przeszukań) do rozmiaru symulacji. Na rysunku 8. przedstawiono zwróconą przez algorytm ścieżkę, dla przypadku, gdy pomiędzy punktami znajduje się przeszkoda.



Rysunek 8: Przedstawienie działania algorytmu RRT* w symulatorze (źródło: opracowanie własne

2. Część 2 - Implementacja systemu

2.1.

2.2.

2.3.

3. Cześć 3 - Wykonanie eksperymentów symulacyjnych

3.1.

3.2.

3.3.