POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Automatyka i Robotyka
SPECJALNOŚĆ: Technologie informacyjne w systemach automatyki

PRACA DYPLOMOWA INŻYNIERSKA

Symulator robotów mobilnych w narzędziu Unity

Mobile robot simulator in the Unity tool

AUTOR:
Jakub Mateusz Szczyrk

PROWADZĄCY PRACĘ:
dr inż. Piotr Sobolewski KSSK

OCENA PRACY:

WROCŁAW, 2019

Spis treści

Spis rysunków

Listings

Rozdział 1

Cel i zakres pracy

Celem przedstawianej pracy inżynierskiej było stworzenie symulatora robotów mobilnych umożliwiającego użytkownikowi sprawdzenie ich algorytmu na własnym trójwymiarowym modelu robota w samodzielnie zaprojektowanym środowisku. Symulator ten został wygenerowany w narzędziu Unity, a środowisko oraz robot importowane są w rozszerzeniu .fbx do symulatora. Robota poddaje się edycji, gdzie należy dokonać wyboru, które elementy to koła i sensory. Urządzenia te potrzebują wprowadzenia parametrów pozwalających prawidłowo odzwierciedlić rzeczywistość. Symulator stara się sam prawidłowo wyznaczyć promień koło, lecz nie zawsze jest to możliwe ze względu na niepoprawny model robota. Do wprowadzenia algorytmu służy specjalne pole tekstowe, sam algorytm jest pisany w języku C#, a zmienne dla kół i czujników są wyświetlane na górze pola tekstowego. Zmienna dla koła przedstawia moc silników, natomiast zmienna dla czujnika informuje czy w polu widzenia znajduję się jakiś obiekt. Symulator umożliwia trzy tryby używania kamery: widok z góry, swobodne poruszanie się i orbitowanie wokół robotów. Dodatkowe funkcje edycji symulatora osiąga się przy pomocy edytora Unity.

Praca składa się z sześciu rozdziałów poruszających różne kategorie. We wstępie wprowadzone są pojęcia związane z robotyką. W rozdziale trzecim opisane są narzędzia oraz technologię użyte w pracy. Rozdział czwarty zawiera opis projektu oraz jego działanie i sposób użycia. W rozdziale piątym opisane zostały najważniejsze użyte komponenty środowiska Unity oraz kod programu. Rozdział ostatni podsumowuje całą pracę, wyjaśnia dlaczego projekt jest uruchamiany w Unity oraz przytacza przykłady dalszego rozwoju aplikacji.

Rozdział 2

Wstęp

Urządzenie można nazwać robotem, dokładnie opisanym w artykule [14], gdy zawiera ono ruchomy mechanizm, na który wpływ mają komponenty wykrywające, planujące, uruchamiające i sterujące. Jednak minimalna liczba tych komponentów nie musi być zaimplementowana w oprogramowaniu lub być możliwa do zmiany przez konsumenta, który używa urządzenia, czyli zachowanie ruchowe może zostać na stałe podłączone do urządzenia przez producenta. Robot jest urządzeniem zdolnym do wykonywania czynności jak człowiek, posiada programowalny manipulator, który może wykonywać wiele operacji zgodnie z zaprogramowanymi ścieżkami, aby spełnić różnorodne zadania. W aspekcie funkcjonalnym w robocie wyszczególnia się układ mechaniczny, układ sterowania oraz układ zasilania. Układ sterowania określa sygnał sterowania, który należy podjąć wobec systemu (robota), aby otrzymać z góry założone właściwości. Sygnał sterujący generowany jest na podstawie posiadanych danych o tym systemie, a w przypadku robota zadaniem układu sterowania jest takie generowanie sygnałów sterujących, aby układ osiągnął żądaną pozycję i orientację w przestrzeni z uwzględnieniem omijania przeszkód i kontrolowaniem podstawowych parametrów kinematycznych i dynamicznych. Sterowanie takim robotem polega na zapewnieniu określonego następstwa kroków programu działania i zapewnieniu właściwej realizacji poszczególnych kroków działania.

Mobilnym robotem nazywa się platformę o dużej mobilności w swoim środowisku (powietrznym, lądowym, podwodnym), będącą systemem o cechach funkcjonalnych takich jak mobilność (całkowita mobilność w stosunku do środowiska), pewien poziom autonomii (ograniczona interakcja człowieka) oraz zdolność percepcji (wykrywanie i reagowanie w otoczeniu). Mobilne przynależą do klasy robotów, które mogą się przemieszczać za pomocą kół lub gąsienic, a poprzez otoczenie robota rozumie się przestrzeń, w której robot jest usytuowany. Dla robotów stacjonarnych otoczenie ogranicza się do przestrzeni roboczej.

Robotyka w dużym stopniu polega na integracji systemu, osiągnięciu zadania przez uruchamiane urządzenie mechaniczne, poprzez inteligentną integrację komponentów, z których wiele dzieli z innymi dziedzinami, takimi jak systemy i kontrola, informatyka, animacja postaci, projektowanie maszyn, sztuczna inteligencja. Zatem granice robotyki nie mogą być jasno określone, ponieważ coraz więcej jej pomysłów, koncepcji i algorytmów stosuje się w większej liczbie aplikacji zewnętrznych i odwrotnie, podstawowe technologie z innych dziedzin (na przykład wizja, biologia, kognitywistyka lub biomechanika) stają się kluczowymi komponentami w coraz bardziej nowoczesnych systemach robotycznych.

Robotyka mobilna, omówiona szczegółowo w książce [20], związana jest z tworzeniem robotów mobilnych, które mogą poruszać się w środowisku fizycznym. Roboty mobilne są zwykle kontrolowane przez oprogramowanie i używają czujników oraz innego sprzętu do identyfikowania swojego otoczenia łącząc postęp w sztucznej inteligencji z robotyką fizyczną, która pozwala im poruszać się po otoczeniu. Robotyka mobilna dzieli się na dwa rodzaje - autonomiczne

i nieautonomiczne roboty mobilne. Autonomia robota wiąże się ze zdolnością samodzielnego osiągania zadanego celu, a mobilność to jego zdolność do przemieszczania się w całości w otaczającym środowisku. Autonomiczne roboty mobilne, których wszystkie elementy zostały zintegrowane w ramach konstrukcji mechanicznej, mogą eksplorować otoczenie bez zewnętrznego prowadzenia, podczas gdy roboty kierowane używają pewnego rodzaju systemu prowadzenia do poruszania się. Inne roboty półstacjonarne mają niewielki zakres ruchu. Robot mobilny stanowi połączenie różnych komponentów fizycznych (sprzętowych) i obliczeniowych (programowych). Pod względem komponentów sprzętowych można rozważyć robota mobilnego jako zbiór podsystemów - jak robot porusza się w swoim otoczeniu, jak mierzy właściwości siebie i swojego środowiska, jak mapuje pomiary na działania, jak komunikuje się z otoczeniem. Roboty mobilne autonomiczne mogą poruszać się w niekontrolowanym środowisku bez potrzeby stosowania fizycznych lub elektromechanicznych urządzeń naprowadzających, natomiast roboty mobilne polegają na urządzeniach nawigacyjnych, które umożliwiają im podróżowanie z góry określoną trasą nawigacyjną w stosunkowo kontrolowanej przestrzeni.

W skład robota mobilnego wchodzą sterownik, oprogramowanie sterujące, czujniki i urządzenia wykonawcze. Kontroler jest generalnie mikroprocesorem, wbudowanym mikrokontrolerem lub komputerem osobistym (PC). Mobilne oprogramowanie sterujące może być językiem asemblera lub językiem wysokiego poziomu, takim jak C, C ++, Pascal, Fortran lub specjalnym oprogramowaniem w czasie rzeczywistym.

Zastosowane czujniki zależą od wymagań robota, które mogą obejmować obliczenia martwe, wykrywanie dotykowe i zbliżeniowe, określanie zasięgu triangulacji, unikanie kolizji, lokalizację pozycji i inne szczególne zastosowania. Symulator robotyki to symulator służący do tworzenia aplikacji dla robota fizycznego bez zależności od rzeczywistej maszyny, co pozwala zaoszczędzić koszty i czas. W niektórych przypadkach aplikacje te można przenieść na robota fizycznego (lub przebudować) bez modyfikacji. Robot mobilny musi osiągnąć swój cel w bardzo złożonych środowiskach z niepewnością czujników i urządzeń wykonawczych. Z powodu takich niepewności algorytm sterowania zachowaniem robota musi być w stanie poradzić sobie z różnymi możliwymi sytuacjami środowiskowymi i statusem robota. Aby opracować taki algorytm sterowania zachowaniem robota, algorytm musi zostać przetestowany w wielu warunkach otoczenia robota. Taki proces wymaga dużej liczby eksperymentów z wykorzystaniem prawdziwych robotów, a ze względu na wysokie koszty eksperymentalne i złożoność środowiska należy opracować realistyczny symulator do weryfikacji algorytmów zachowania.

W aplikacjach robotyki mobilnej symulatory robotyki, działające w oparciu o zachowanie, umożliwiają użytkownikom tworzenie prostych światów sztywnych obiektów i źródeł światła oraz programowanie robotów do interakcji z tymi światami. Symulacja, przedstawiona w książce [20], bazująca na zachowaniu pozwala na działania o charakterze bardziej biologicznym w porównaniu do symulatorów, które są bardziej binarne lub obliczeniowe. Co więcej symulatory oparte na zachowaniu mogą „uczyć się” na błędach i są w stanie wykazać antropomorficzną jakość wytrzymałości. Często spotykaną aplikacją do symulatorów robotyki jest modelowanie 3D i generowanie robota wraz z jego środowiskiem. Niektóre symulatory robotyki wykorzystują silnik fizyki do bardziej realistycznego generowania ruchu robota. Zastosowanie symulatora robotyki do opracowania programu kontroli robotyki jest wysoce zalecane bez względu na to, czy rzeczywisty robot jest dostępny, czy też nie. Symulator pozwala na wygodne pisanie i debugowanie programów robotyki w trybie offline z ostateczną wersją programu testowanego na rzeczywistym robocie. Dotyczy to głównie robotycznych aplikacji przemysłowych, ponieważ powodzenie programowania offline zależy od tego, jak podobne jest rzeczywiste środowisko robota do środowiska symulowanego.

Rozdział 3

Narzędzia i technologie

3.1. UNITY

Unity to wieloplatformowy silnik gry, który został szczegółowo opisany w książce [18]. Unity koncentruje się na rozwoju aplikacji, w szczególności gier 2D i 3D oraz treści interaktywnych, a obecnie obsługuje ponad 20 różnych platform docelowych do wdrażania z których najpopularniejsze platformy to komputery PC, Android i iOS. Unity zawiera kompletny zestaw narzędzi do projektowania i budowania symulacji, w tym interfejsy do grafiki, dźwięku i narzędzi do budowania środowiska, wymagające minimalnego użycia zewnętrznych programów do pracy nad projektami. Po utworzeniu projektu pojawia się pierwszy plik ze sceną. Po lewej stronie znajduje się sekcja Hierarchy, scharakteryzowana w dokumentacji [8], czyli hierarchia obiektów w grze. Na początku na scenie obecne są dwa obiekty. Pierwszy z nich – Main Camera to obiekt, dokładnie opisany w dokumentacji [2], który pełni rolę wzroku użytkownika. Tworząc projekt i przemieszczając pozycję Main Camera, tak naprawdę porusza się kamerą, zmieniając obraz, który jest w danym momencie widoczny w oknie Game, zdefiniowanym w dokumentacji [5]. Drugi obiekt to Directional Light, przedłożony w dokumentacji [9], czyli światło, które służy do odpowiedniego oświetlenia obiektów sceny i jest wykorzystywane podczas generowania projektu. Po prawej stronie umiejscowiona jest sekcja Inspector, ukazana w dokumentacji [16], będąca odpowiednikiem sekcji Właściwości w MS Visual Studio, która odpowiedzialna jest za wyświetlanie właściwości zaznaczonych obiektów. Pomiędzy hierarchią a inspektorem widoczny jest podgląd sceny na którym widać wszystkie dodawane obiekty. Na dole okna programu znajduje się sekcja Project, w której znajduje się menedżer zasobów projektu pozwalający na umieszczanie obiektów na scenie. Podgląd sceny to model trójwymiarowy, zatem scenę możemy oglądać z różnych perspektyw:

- Przybliżanie i oddalanie – obracanie kółkiem myszy.
- Przesuwanie – ruch kursora przy wciśniętym kółku myszy.
- Obracanie – ruch kursora przy wciśniętym prawym przycisku myszy.

Widok Sceny to miejsce, w którym tworzy się projekt, dodaje się wszystkie modele, kamery i inne elementy, dokładnie zademonstrowane w dokumentacji [12]. To okno 3D, w którym można wizualnie umieścić wszystkie używane zasoby. Podczas testowania projektu w Unity widok sceny aktualizowany jest do aktualnego stanu. Po zatrzymaniu aplikacji widok sceny powróci do pierwotnego stanu. Wszelkie zmiany wprowadzone w widoku Sceny podczas działania programu zostaną utracone po zatrzymaniu. Widok Game przedstawia perspektywę użytkownika, można w tym miejscu przetestować program i sprawdzić jak działają różne mechanizmy. Okno Game posiada selektor współczynnika kształtu, który umożliwia zmianę wymiarów widoku w celu dopasowania do wymiaru określonego lub rozmiaru ekranu urządzenia, takiego jak

iPhone'a lub iPada. Pozwala to upewnić się, że aplikacja wygląda dobrze we wszystkich proporcjach i rozdzielczościach, które ma obsługiwać oraz że żadna ważna zawartość nie jest odcięta. GameObjects, klarownie wytłumaczone w dokumentacji [4], to puste pojemniki, które można dostosowywać, dodając składniki. Komponenty pozwalają GameObjects rzutować geometrię, emitować światło, działać jak kamera, a nawet tworzyć złożone zachowania za pomocą skryptów. GameObjects mogą także zachowywać się jak foldery zawierające inne GameObjects, co czyni je bardzo przydatnymi do organizowania sceny. Scena zazwyczaj reprezentuje jeden poziom tworzonej gry, mimo, że teoretycznie można umieścić całą grę w jednej scenie. Wszelkie obiekty GameObjects aktywnie używane w aplikacji w bieżącej scenie pojawiają się w oknie Hierarchia. Każda nowa scena zaczyna się od kamery głównej i światła kierunkowego, które są jednocześnie obiektami GameObjects. Należy pamiętać, że nie trzeba ich zatrzymywać, zawsze można je usunąć i dodać później. Każde usunięcie GameObject z Hierarchi usuwa go ze sceny. Gdy scena rośnie z dużą ilością GameObjects, ważną rolę odgrywa pasek wyszukiwania. Okno projektu zawiera wszystkie zasoby używane w aplikacji. Można uporządkować wszystkie zasoby według folderów, a w momencie, gdy pojawia się konieczność skorzystania z któregoś z nich przeciąga się te zasoby z okna Projekt do okna Hierarchia. Istnieje także możliwość przeciągnięcia zasobów z okna Projekt do widoku Scena, a jeśli przeciągnie się pliki z komputera do okna projektu, Unity automatycznie zaimportuje je jako zasoby. Unity organizuje zasoby w oknie projektu tak samo, jak system plików OS, utrzymuje metadane dla każdego zasobu, więc przenoszenie zasobów między folderami w systemie plików psuje metadane. Jeśli należy wprowadzić zmiany organizacyjne do zasobów trzeba wykonywać takie zmiany w oknie Projekt. Okno Inspektora umożliwia skonfigurowanie dowolnego GameObject. Po wybraniu GameObject w Hierarchii, Inspektor wyświetli listę wszystkich składników GameObject i ich właściwości. Pasek narzędzi służy do manipulowania różnymi obiektami GameObject w widoku Scena. Nawigacja po scenie odbywa się najczęściej za pomocą myszy. Trzy najważniejsze manewry nawigacyjne to przeciąganie, orbitowanie oraz przybliżanie i oddalanie. Wszystkie trzy rodzaje ruchu opierają się na technice „kliknij i przeciągnij" z jednoczesnym naciśnięciem pewnej kombinacji klawiszy Alt i Ctrl w zależności od typu myszy. Przesunięcie odpowiada translacji kamery, orbitowanie to rotacja kamery, natomiast przybliżanie i oddalanie to skalowanie kamery.

Unity to oparty na komponentach modułowy system przeznaczony do budowy obiektów w aplikacjach, gdzie komponent oznacza pakiet funkcjonalności przybliżonych na stronie internetowej [15], a obiekty w aplikacji budowane są jako kolekcje komponentów, nie bazując na ścisłej hierarchii klas. W unity obiekty istnieją na płaskiej płaszczyźnie, gdzie poszczególne obiekty mogą mieć różne kolekcje komponentów. Różni się to od podejścia opartego na strukturze dziedziczenia, w którym poszczególne obiekty znajdują się na różnych gałęziach drzewa. Takie rozwiązanie ułatwia sprawne tworzenie prototypów, gdyż daje możliwość szybkiego dopasowywania różnych komponentów, zamiast wymaganej refaktoryzacji łańcucha dziedziczenia w przypadku zmiany obiektów. Zaletą Unity jest również bardzo produktywny wizualny sposób pracy oraz wysoki stopień niezależności od platformy. Wizualny sposób pracy jest dużym ułatwieniem dzięki istnieniu zaawansowanego edytora graficznego. Edytor ten pozwala na przygotowanie sceny w aplikacji i na powiązanie zasobów graficznych oraz kodu w interaktywnych obiektach, napisanego w $C\#$, który został omówiony w artykule [19]. Można dostosowywać obiekty w edytorze, przeprowadzać na nich operacje w momencie, gdy aplikacja jest uruchomiona oraz dostosowywać do własnych potrzeb sam edytor poprzez skrypty dodające nowe funkcje do menu i interfejsu użytkownika. Niezależność od platformy przejawia się nie tylko wybór dowolnej platformy docelowej dla tworzonej aplikacji, ale także w katalogach narzędzi programistycznych, ponieważ grę można opracować w systemie Windows lub macOS. Niewiele silników aplikacji obsługuje tak dużą liczbę docelowych platform wdrożenia, a Unity zapewnia najłatwiejsze wdrożenie na wielu platformach.

W widoku Scene można umieszczać obiekty i poruszać nimi, zobaczyć pełną liczbę obiektów przedstawianych za pomocą różnych ikon i kolorowych linii, takich jak kamery, światła, obszary kolizji. Widoczny tu widok nie jest taki sam jak w uruchamianej symulacji, ponieważ można rozglądać się po scenie bez ograniczeń nakładanych na widok symulacji. W skład scen wchodzą środowiska i menu aplikacji. Można powiedzieć, że każdy unikalny plik Scene to unikalny poziom. W każdej scenie umieszcza się środowiska, przeszkody i dekoracje, zasadniczo projektując i budując aplikację w kawałkach. Podczas tworzenia nowego projektu Unity, widok sceny wyświetla nową scenę. Ta scena nie posiada tytułu, nie jest zapisana, jest pusta, z wyjątkiem kamery głównej i światła kierunkowego. W przeciwieństwie do innych obiektów tworzonych w panelu Projekt, sceny są tworzone za pomocą menu Plik. Sceny należy zapisywać ręcznie, a klasycznym błędem w Unity jest wprowadzanie zmian w scenie i jej elementach oraz zapominanie o zapisaniu jej później. Narzędzie kontroli wersji nie zobaczy żadnych zmian, dopóki scena nie zostanie zapisana.

3.2. MICROSOFT VISUAL STUDIO

Microsoft Visual Studio, zreferowane w artykule [13], to zintegrowane środowisko programistyczne firmy Microsoft stosowane do tworzenia oprogramowania konsolowego oraz z graficznym interfejsem użytkownika. W skład Microsoft Visual Studio wchodzi edytor kodu wspierający IntelliSense jak również mechanizmy refaktoryzacji kodu, zintegrowany debugger, który działa zarówno na poziomie kodu źródłowego jak i maszyny, designer do tworzenia aplikacji Windows Forms, WPF i web, narzędzie do tworzenia klas, projektowania baz danych itd. Microsoft Visual Studio umożliwia edytowanie, debugowanie i kompilowanie kodu, a następnie publikowanie aplikacji. Eksplorator rozwiązań umożliwia wyświetlanie plików kodu i nawigowanie w nich oraz zarządzanie nimi oraz pomaga organizować kod przez zgrupowanie plików na rozwiązania i projekty. Okno edytora, wyświetla zawartość pliku, jest to miejsce, w którym można edytować kod lub zaprojektować interfejs użytkownika, taki jak okno z przyciskami i polami tekstowymi. Team Explorer pozwala na śledzenie elementów roboczych i udostępnianie kodu innym osobom korzystającym z technologii kontroli wersji, takich jak git i kontrola wersji serwera Team Foundation (TFVC). Użycie Visual Studio Code z rozszerzeniem C#, zapewnia zaawansowane środowisko edycji z pełną obsługą C# technologii IntelliSense (inteligentnych uzupełniania kodu) i debugowania.

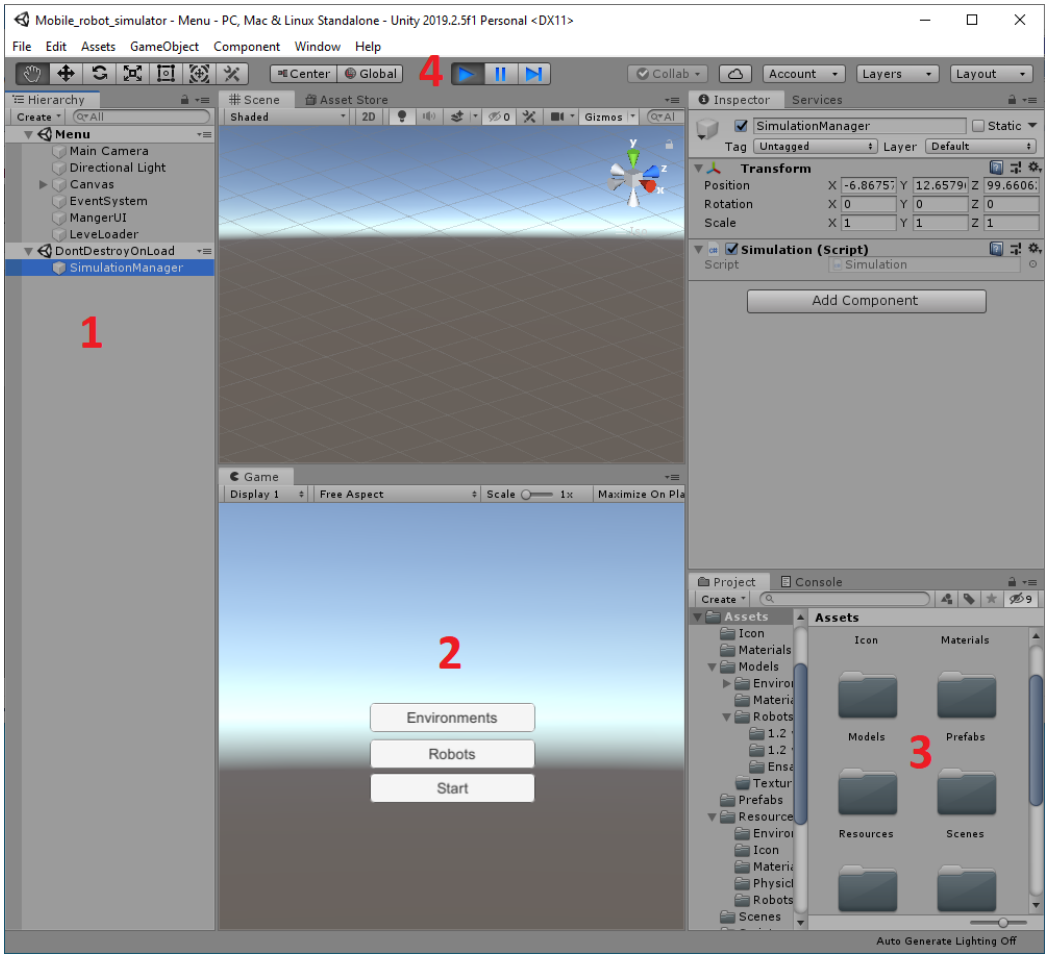
3.3. Github

Git, opisany w artykule [6], to system kontroli wersji, czyli oprogramowanie służące do śledzenia zmian głównie w kodzie źródłowym oraz pomocy programistom w łączeniu zmian dokonanych w plikach przez wiele osób w różnym czasie. Natomiast GitHub, przedstawiony na stronie internetowej [7], to hostingowy serwis internetowy przeznaczony dla projektów programistycznych wykorzystujących system kontroli wersji Git. Zatem Git jest narzędziem, a Github usługą w której można przechowywać projekty korzystające z Git. Commit to indywidualna zmiana w pliku (lub zestawie plików), zwykle zawiera komunikat zatwierdzenia, który jest krótkim opisem wprowadzonych zmian. Repozytorium jest najbardziej podstawowym elementem Github i można je porównać do folderu projektu. Repozytorium zawiera wszystkie pliki projektu (w tym dokumentację) i przechowuje historię zmian każdego pliku, może mieć wielu współpracowników, być publiczne lub prywatne. Pull odnosi się do pobierania i łączenia zmian, a push oznacza wysyłanie zatwierdzonych zmian do zdalnego repozytorium, takiego jak repozytorium hostowane w Github.

Rozdział 4

Projekt

Projekt działa w Unity, gdzie można wyszczególnić kilka podstawowych obszarów roboczych, zgodnie z podziałem pokazanym na rysunku 4.1. Pierwszym z nich jest Hierachy, który zawiera elementy znajdujące się na scenie Menu umiejscowionej w /Assets/Scenes i sceny DontDestroyOnLoad tworzącej się podczas startu. Scena DontDestroyOnLoad posiada SimulationManager odpowiadający za główne działanie aplikacji. Drugi obszar roboczy to widok symulacji. Trzecim wyszczególnionym miejscem jest przeglądarka plików Project, można tu przełączać widoczną scenę, dodawać elementy do sceny – roboty, przeszkody. Czwarty obszar to przyciski Play / Pause / Step pozwalające na uruchomienie sceny w widoku Game i start symulacji.



Rys. 4.1: Projekt w Unity

4.1. Scena Menu

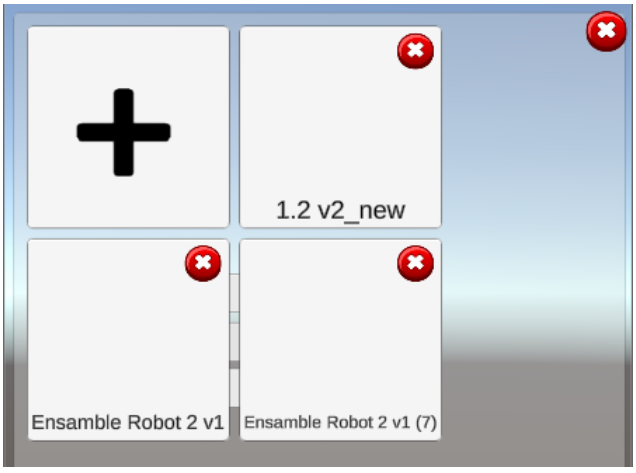
Odpowiada za dodawanie robotów i środowisk do Unity. Po kliknięciu Play, w widoku Game ukaże się menu wraz z przyciskami:

- Environments – służącym do dodawania oraz wybierania środowiska dla robota
- Robots – umożliwiającym dodawanie i wybieranie robotów, które zostaną poddane symulacji
- Start – pozwalającym na przejście do sceny Simulator

Panel dodawania środowiska zaprezentowany został na rysunku 4.2. Po wciśnięciu przycisku Environments można dodać środowisko (+), wybrać lub usunąć istniejącą już arenę. Klikając biały krzyżyk na czerwonym tle przy środowisku usuwamy go z folderu Assets/Resources/Environments. Wybierając dane środowisko poprzez kliknięcie np. Arena MiniSumo, zostanie ono utworzone w następnej scenie.

Rys. 4.2: Panel dodawania środowiska

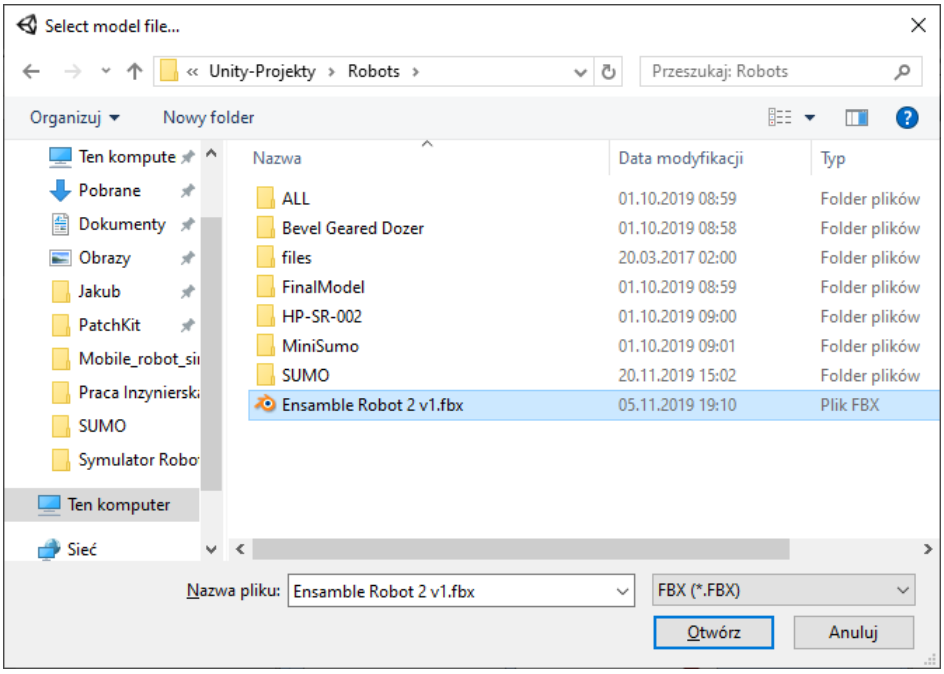
Rysunek 4.3 ilustruje panel dodawania robota. Naciskając przycisk Robots można dodać robota (+), wybrać lub usunąć istniejący już robot. Klikając biały krzyżyk na czerwonym tle przy robocie usuwamy go z folderu Assets/Resources/Robots. Wybierając dany robot poprzez wciśnięcie np. Ensamble Robot 2 v1, zostanie on utworzony w następnej scenie.



Rys. 4.3: Panel dodawania Robota

Zarówno przy Environments jak i Robots, klikając krzyżyk w prawym górnym rogu okna aplikacji zamyka się panel odpowiednio dla środowiska lub robotów.

Przycisk „+” otwiera okno dialogowe wyboru pliku, odpowiadające za dodawanie modelu o rozszerzeniu FBX przedstawione na rysunku 4.4. FBX to otwarty i niezależny od platformy format pliku danych 3D należący do Autodesk. Pliki tego typu przechowują rysunki 2D lub 3D z zachowaniem wierności i funkcjonalności oryginalnego pliku. Format FBX zapewnia kompatybilność danych między różnymi aplikacjami do tworzenia treści cyfrowych, ponadto pliki FBX pozwalają użytkownikom tworzyć i edytować dane cyfrowe za pomocą wielu różnych narzędzi do edycji 3D, niezależnie od platformy systemowej. Po wybraniu modelu tworzony jest Prefab znajdujący się w Assets/Resources/Robots, w którym dodane są Collider oraz skrypty wykorzystywane w symulacji i edycji.



Rys. 4.4: Okno dialogowe wyboru pliku .FBX

4.2. Scena Simulator

Scena uruchamia się poprzez kliknięcie Start w scenie Menu lub wybranie z przeglądarki plików (Project) pliku Assets/Senes/Simulator. Dodanie elementów do sceny odbywa się za pomocą:

- Wybranych elementów w scenie Menu
- Aktywowania obiektów na scenie przy użyciu Hierarchy
- Przeciągnięć obiektów z przeglądarki plików (Project) do Hierarchy lub na okno Scene (czasami nie jest to możliwe)

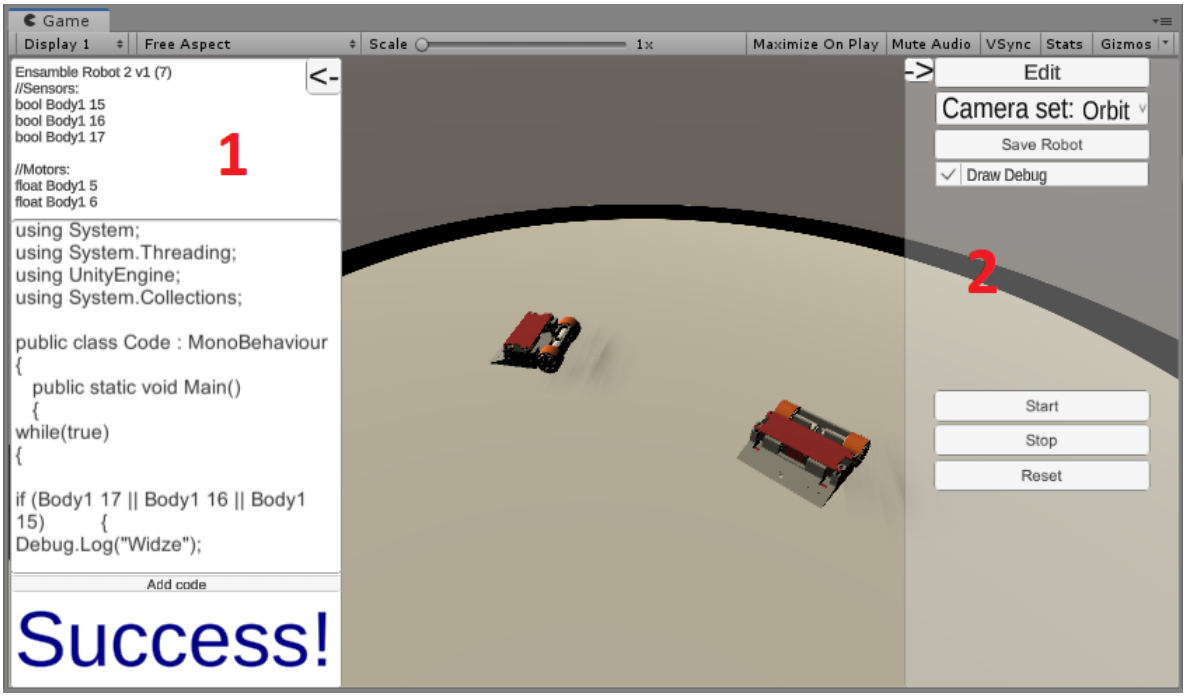
W oknie Game zilustrowanym rysunkiem 4.5 mamy następujące panele:

- Panel pierwszy:
  - Informacje o nazwie zaznaczonego robota oraz zmienne używane w kodzie.
  - Pole tekstowe, gdzie wprowadza się algorytm dla robota. Raz wprowadzony kod jest zapisywany i implementowany do każdego robota przy starcie symulacji.
  - Przycisk „Add code” umożliwiający wprowadzenie programu do robota.

  - Pole tekstowe z informacją o błędzie lub poprawnym wgraniu się programu poprzez komunikat „Success!” lub wylistowanie na czerwono błędów w kodzie.

- Panel drugi:
  - Przycisk „Edit” otwierający edycję robota.
  - Lista rozwijana Camera set pozwalająca na wybór jednego z 3 typów widoków: birdseye, free movement i orbit.
  - Przycisk „Save Robot” zapisujący zmiany dokonane w robocie.
  - Draw debug – zaznaczony pokazuje pole widzenia.
  - Przycisk „Start” uruchamiający symulację robota, „Stop” zatrzymujący ją oraz „Reset” ustawiający robota w pozycji wyjściowej.

W panelach widoczne są przyciski <- lub ->, które pozwalają na chowanie i pokazywanie się panelu



Rys. 4.5: Scena Simulator

4.2.1. Edycja robota

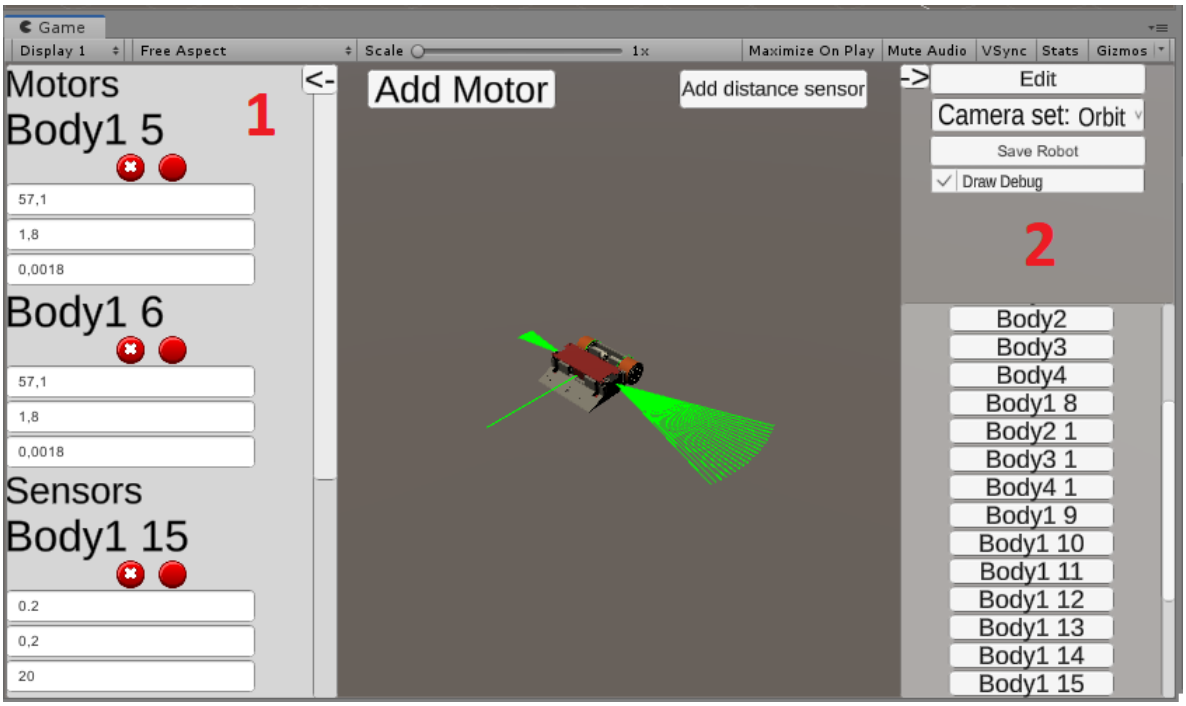
Edycja robota pokazana na rysunku 4.6 polega na dodawaniu, usuwaniu lub zamianie parametrów silników bądź czujników. W Unity za symulowanie silnika odpowiada wheel Collider dodawany razem ze skryptem obsługującym silnik do koła robota, a nie do oryginalnego silnika robota. Silnik i sensory posiadają parametry, które pozwalają dobrze zasymulować realistyczny obiekt. Aby zobrazować działanie sensora wyświetlane są zielone linie, które ograniczają pole widzenia. Symulator automatycznie próbuje prawidłowo wyznaczyć kierunek oraz działanie czujnika, ale nie zawsze się to udaje. Należy wtedy w Hierarchy znaleźć obiekt „Start Light”, który za rodzica będzie miał czujnik robota. GameObject (5.1.1) ten trzeba prawidłowo obrócić i ustawić pozycję początkową promieni czujnika. Wszystkie zmiany na tym obiekcie będą widoczne w widoku Game poprzez zielone linie.

Silniki mają wyświetlane zielone okręgi, które pomagają ustawić promień oraz środek koła. W przypadku, gdy w widoku Game środek koła pokazuje się w złym miejscu, należy sprawdzić

poprawność jego położenia w widoku Scene. W Hierarchy po nazwie silnika można znaleźć koło w którym będzie wheel Collider. W tym komponencie istotne są dwa parametry radius i center, które należy prawidłowo ustawić, a zmiany te będą widoczne w widoku Scene poprzez zielone okręgi.

Elementy robota podczas edycji można zaznaczać lewym przyciskiem myszki. Zmianie ulega wtedy ich kolor, który przyjmuje barwę fioletową. Po od kliknięciu kolor powraca do stanu początkowego. Po wybraniu konkretnego elementu w centrum widoku pojawiają się dwa przyciski – „Add Motor" dodający silnik oraz „Add distance sensor" dodający czujnik odległościowy – do panelu pierwszego i edytowanego robota.

Pierwszy panel zawiera dwie sekcje: motors, czyli silniki oraz sensors – czujniki odległościowe. W sekcji motors wprowadza się kolejno maksymalne obroty na minutę, maksymalny moment obrotowy i promień koła. W sekcji sensors należy podać maksymalny zasięg, częstotliwość oraz kąt widzenia. Biały krzyżyk na czerwonym tle służy do usuwania konkretnego silnika lub sensora. Czerwona kropka zaznacza element na edytowanym robocie. Drugi panel obsługuje zaznaczanie poszczególnych części robota poddawanych edycji, poprzez kliknięcie w przycisk z nazwą elementu robota.



Rys. 4.6: Edycja robota

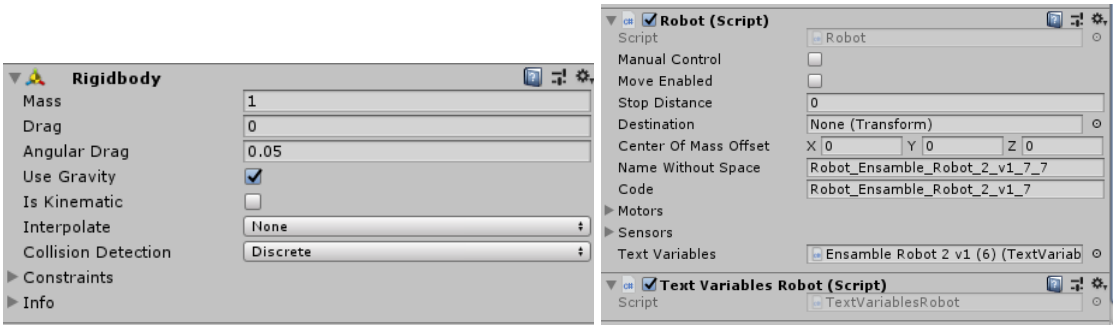
4.2.2. Tryby kamery

Symulator, celem ułatwienia użytkownikowi obserwowanie robotów w widoku Game, posiada trzy tryby kamery:

- Orbit – pozwala na orbitowanie kamerą wokół zaznaczonego robota za pomocą prawego przycisku. Umożliwia także przybliżanie lub oddalanie widoku przy użyciu kółka przewijania.
- Free Movment – pozwala na samodzielne poruszanie się kamerą za pomocą strzałek lub W,A,S,D , a poprzez prawy przycisk myszki obraca kamerę
- Bridseye – pozwala na patrzenie z góry na zaznaczonego robota, a za pomocą kółka przewijania można przybliżać lub oddalać widok

4.2.3. Robot

Robotem nazywamy obiekt poddawany symulacji, który posiada skrypt Robot. Jest on w hierarchy rodzicem dla wszystkich elementów robota, w tym sinika i czujnika odległościowego. Dodane w scenie Menu roboty znajdują się w Assets/Resources/Environments i przy pomocy przeciągania prefabów na scenę z przeglądu plików (Project) dodawane są do symulacji. Robot symulowany jest poprzez komponent Rigidbody, opisany w dokumentacji [11], zamieszczony na rysunku 4.7. Pozwala to na prawidłową reakcję robota na grawitację. Istotnym parametrem jest masa oraz środek masy robota. Masę można zmienić w Rigidbody. Środek masy jest wyznaczany przez Unity, ale można go zmodyfikować zmieniając parametr Center Of Mass Offset w Robot.



Rys. 4.7: Komponenty do symulowania robota

Każdy robot może posiadać swój indywidualny program napisany w języku c# i wprowadzany w symulacji. Kod jest zapisywany w głównym folderze projektu z nazwą zaczynającą się od Robot_, posiadającą nazwę robota i rozszerzeniem .txt. Analogicznie tworzone są pliki .dll, które wykorzystuje symulator. Do sterowania silnikami użytkownik wykorzystuje nazwę silnika i wprowadza go w następujący sposób, dla sinika o nazwie "Body1 5":

- Body1 5 = 20; - koło będzie się kręcić do przodu
- Body1 5 = 0; - koło nie będzie się kręcić
- Body1 5 = -20; - koło będzie się kręcić do tyłu

Do odbierania informacji o tym, że czujnik odległościowy wykrył obiekt w polu widzenia, wykorzystuje się nazwę sensora: if(Body1 17) - jeżeli czujnik wykryje obiekt zosatnie wykonany kod pod instrukcją warunkową if.

W przypadku gdy robot nie ma wprowadzonego kodu, pokazuje się przykładowy kod:

Listing 4.1: Przykładowy kod.

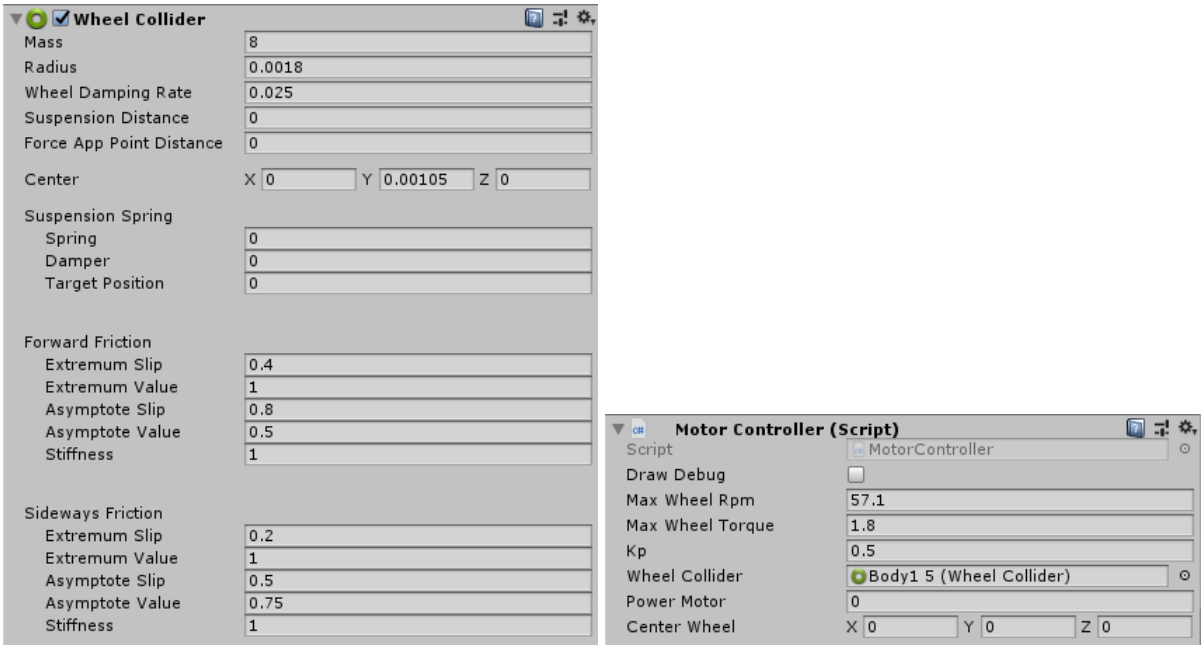
```
using System;
using System.Threading;
using UnityEngine;
using System.Collections;

public class Code : MonoBehaviour  {
public static void Main()
{
        while(true)
        {
                if ([bool sensor]){
                [float motor1]=[value(-20:20)];
                [float motor2]=[value(-20:20)];}
        else{
                [float motor1]=[value(-20:20)];
```

```
                [float motor2]=[value(-20:20)];}
        Thread.Sleep(500);
    }
}}
```

4.2.4. Silnik

Sinik symulowany jest poprzez komponent Wheel Collider, przedłożony w dokumentacji [17], zobrazowany rysunkiem 4.8. Oprócz wspomnianych w rozdziale edycji robota parametrów center i radius, mamy także parametry Mass i Wheel Damping Rate. Zmiany w nich mogą zmienić poruszanie się symulowanego robota.

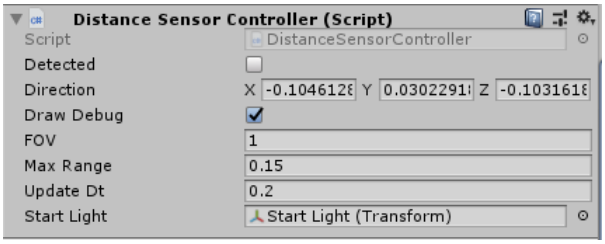


Rys. 4.8: Komponenty do symulowania silnika

4.2.5. Czujnik odległościowy

Czujnik symulowany jest poprzez skrypt Distance Sensor Contoller, ukazany na rysunku 4.9. Istotnymi jego parametrami są:

- Fov (Field of view) – pole widzenia które podawane są w stopniach[°]
- Max Range – maksymalmny zasięg, który podawany jest w metrach[m]
- Update Dt – Czas odświeżania się działania sensora podawany w sekundach [s]



Rys. 4.9: Komponent do symulowania czujnika odległościowego

4.2.6. Środowisko

Środowiskiem nazywane są wszystkie elementy posiadające collider poza robotem. Dodane w scenie Menu środowiska znajdują się w Assets/Resources/Environments i przy pomocy przeciągania prefabów na scenę z przeglądu plików (Project) dodawane są obiekty. Unity umożliwia również dodawanie wbudowanych obiektów poprzez zakładkę GameObject z której następnie należy wybrać 3D Object. Obiekty te mogą być podłożem, ścianą lub przeszkodą dla robota. Jeśli robot ma za zadanie przesuwać obiekty umieszczone na scenie, trzeba dodać komponent Rigidbody z ustawieniem odpowiedniej masy. W celu dodania Rigidbody należy wejść w Inspector dla danego obiektu, wcisnąć przycisk „Add Component” i wybrać Rigidbody.

Rozdział 5

Implementacja

5.1. Wykorzystane komponenty Unity

5.1.1. GameObjects

GameObjects to podstawowe obiekty w Unity, które reprezentują postacie, rekwizyty i scenerię oraz działają jak pojemniki na komponenty, które wdrażają prawdziwą funkcjonalność. GameObject to najważniejsza koncepcja w edytorze Unity. Każdy obiekt w grze jest typu GameObject, co oznacza, że wszystko, co można wymyślić w aplikacji musi być typu GameObject. Zanim GameObject stanie się postacią, środowiskiem lub efektem specjalnym, należy nadać mu właściwości. Można zatem powiedzieć, że GameObject to kontener do którego dodaje się elementy jak do pojemnika, aby przekształcić go w postać, światło, drzewo, dźwięk lub cokolwiek innego. Każdy dodany element nazywa się komponentem, a w zależności od rodzaju obiektu, który ma zostać stworzony, dodaje się różne kombinacje komponentów do GameObject. Można wyobrazić sobie GameObject jako pusty garnek do gotowania, a komponenty jako różne składniki, które tworzą daną recepturę. Unity ma wiele różnych wbudowanych typów komponentów oraz istnieje w nim opcja tworzenia własnych komponentów za pomocą interfejsu API skryptów Unity. Wyszczególnia się poniższe właściwości GameObject:

- activeInHierarchy - określa, czy GameObject jest aktywny w scenie
- activeSelf - lokalny stan aktywny tego GameObject. (jedynie odczyt)
- isStatic - tylko interfejs API edytora, który określa, czy obiekt gry jest statyczny
- layer - warstwa, w której znajduje się obiekt gry.
- scene - scena, której częścią jest GameObject.
- tag - znacznik tego obiektu gry.
- transform - transformacja dołączona do tego GameObject.

5.1.2. Rigidbody

Sztywne ciało to główna składowa, która umożliwia zachowanie fizyczne GameObject. Po podłączeniu Rigidbody, obiekt natychmiast zareaguje na grawitację. Ponieważ komponent Rigidbody przejmuje ruch GameObject, do którego jest dołączony, nie należy próbować przenosić go ze skryptu, zmieniając właściwości transformacji, takie jak pozycja i obrót. Zamiast tego powinno się zastosować siły, aby pchnąć GameObject i pozwolić silnikowi fizyki obliczyć wyniki. W niektórych przypadkach zaleca się, aby GameObject miał „sztywne ciało” bez kontrolowania jego ruchu przez silnik fizyki. Zatem postać można kontrolować bezpośrednio z kodu skryptu, jednocześnie zezwalając na wykrywanie go przez wyzwalacze. Ten rodzaj ruchu niefizycznego

wytwarzanego ze skryptu jest znany jako ruch kinematyczny. Składnik Rigidbody ma właściwość o nazwie Is Kinematic, która usuwa go ze sterowania silnikiem fizyki i umożliwia jego kinematyczne przenoszenie ze skryptu. Można zmienić wartość Is Kinematic ze skryptu, aby umożliwić włączanie i wyłączanie fizyki dla obiektu, ale wiąże się to z narzutem wydajności i należy go używać oszczędnie. Gdy ciało sztywne porusza się wolniej niż określona minimalna prędkość liniowa lub obrotowa, silnik fizyki zakłada, że ciało zatrzymało się, a gdy tak się dzieje, GameObject nie poruszy się ponownie, dopóki nie otrzyma kolizji lub siły, dlatego jest ustawiony na tryb „sleeping". Ta optymalizacja oznacza, że procesor nie traci czasu na Rigidbody aż do następnego „przebudzenia" (czyli ponownego uruchomienia). W większości przypadków uśpienie i przebudzenie komponentu Rigidbody przebiega transparentnie, jednak obiekt GameObject może się nie obudzić, jeśli collider statyczny (to znaczy taki bez ciała sztywnego) zostanie do niego przesunięty lub odsunięty poprzez zmianę pozycji transformacji. Może to skutkować zawieszeniem się w powietrzu obiektu Rigidbody GameObject, gdy podłoga zostanie spod niego odsunięta. W takich przypadkach GameObject można obudzić jawnie za pomocą funkcji WakeUp.

5.1.3. Colliders

Komponenty zderzaka, opisane w dokumentacji [3], określają kształt obiektu na potrzeby zderzeń fizycznych. Niewidoczny zderzak nie musi mieć dokładnie takiego samego kształtu jak siatka GameObject - przybliżenie siatki jest często bardziej wydajne i nierozróżnialne w rozgrywce. Najprostsze, najmniej obciążające procesor zderzaki to prymitywne typy zderzaków. W 3D są to Collider Box, Spider Collider i Capsule Collider, a w 2D można stosować Box Collider 2D i Circle Collider 2D, gdzie dowolną ich liczbę można dodać do jednego obiektu, aby utworzyć złożone zderzaki. Zderzaki złożone (compound colliders) mogą często dobrze przybliżać kształt GameObject przy zachowaniu niskiego obciążenia procesora. Można uzyskać większą elastyczność, dodając dodatkowe zderzaki do obiektów potomnych – istnieje możliwość przesuwania ramki względem lokalnych osi obiektu nadrzędnego. Podczas tworzenia takiego zderzaka złożonego należy użyć tylko jednego komponentu Rigidbody, umieszczonego na obiekcie głównym w hierarchii. Są jednak przypadki, w których nawet zderzaki złożone nie są wystarczająco dokładne, dlatego w 3D stosuje się zderzaki siatki (mesh colliders) w celu dokładnego dopasowania kształtu siatki obiektu. Te collidery wymagają znacznie więcej procesora niż prymitywne typy, więc należy używać ich oszczędnie, aby zachować dobrą wydajność. Ponadto zderzak siatki nie jest w stanie zderzyć się z innym zderzakami siatki. W niektórych przypadkach można to obejść, zaznaczając zderzak siatki jako wypukły w inspektorze. Generuje to kształt zderzaka jako „wypukły kadłub", który jest jak oryginalna siatka, ale z wypełnionymi podcięciami. Zaletą tego jest fakt, że wypukły zderzak siatki może zderzać się z innymi zderzakami siatki, dzięki czemu można korzystać z tej funkcji, gdy ma się do czynienia z poruszającą się postacią o odpowiednim kształcie. Jednak dobrą zasadą jest stosowanie zderzaków siatkowych do geometrii sceny i przybliżania kształtu ruchomych obiektów za pomocą złożonych zderzaków prymitywnych. Zderzaki można dodawać do GameObject bez komponentu Rigidbody, aby tworzyć podłogi, ściany i inne nieruchome elementy sceny. Są to tak zwane zderzaki statyczne, których położenia nie wolno zmieniać przy zmianie pozycji transformacji, ponieważ ma to duży wpływ na wydajność silnika fizyki. Zderzaki w obiekcie GameObject, który posiada Rgidbody nazywane są zderzakami dynamicznymi. Zderzaki statyczne mogą wchodzić w interakcje ze zderzakami dynamicznymi, ale w związku z tym, że nie zawierają Rigidbody, nie poruszają się w odpowiedzi na zderzenia.

5.1.4. Wheel Collider

Zderzak kół to specjalny zderzak dla pojazdów z uziemieniem. Ma wbudowane wykrywanie kolizji, fizykę kół i model tarcia opony na poślizgu. Może być stosowany do przedmiotów innych niż koła, ale jest specjalnie zaprojektowany do pojazdów z kołami. Wykrywanie kolizji koła odbywa się poprzez rzut promieniem od środka w dół przez lokalną oś Y. Koło ma promień i może rozciągać się w dół zgodnie z odległością zawieszenia. Pojazd jest sterowany za pomocą skryptów wykorzystując różne właściwości: motorTorque, brakeTorque i steerAngle. Zderzak koła oblicza tarcie oddzielnie od reszty silnika fizyki, wykorzystując model tarcia oparty na poślizgu. Pozwala to na bardziej realistyczne zachowanie, ale także powoduje, że zderzaki kół ignorują standardowe ustawienia materiału fizycznego. Ponieważ pojazdy mogą osiągać duże prędkości, bardzo ważna jest poprawna geometria kolizji toru wyścigowego. W szczególności siatka kolizyjna nie powinna mieć małych wypukłości lub wgnieceń, które tworzą widoczne modele np. słupy ogrodzeniowe. Zwykle siatka kolizyjna dla toru wyścigowego jest wykonywana oddzielnie od widocznej siatki, dzięki czemu siatka kolizyjna jest możliwie gładka. Właściwości zderzaków kół i ich funkcje prezentują się następująco:

- Mass - masa koła
- Radius - promień koła
- Wheel Damping Rate - wartość tłumienia przyłożonego do koła
- Suspension Distance - maksymalna odległość wysunięcia zawieszenia koła, mierzona w przestrzeni lokalnej (zawieszenie zawsze rozciąga się w dół przez lokalną oś Y)
- Force App Point Distance - parametr określający punkt przyłożenia sił koła (oczekuje się, że będzie to w metrach od podstawy koła w pozycji spoczynkowej wzdłuż kierunku jazdy zawieszenia)
- Center - środek koła w lokalnej przestrzeni obiektu.
- Suspension Spring - zawieszenie próbuje osiągnąć pozycję docelową przez dodanie sił sprężyny i tłumienia
- Spring - siła sprężyny próbuje osiągnąć pozycję docelową, a większa wartość tego parametru powoduje, że zawieszenie szybciej osiąga pozycję docelową
- Damper - zmniejsza prędkość zawieszenia, a większa wartość tego parametru powoduje spowolnienie ruchu sprężyny zawieszenia.
- Target Position - odległość odpoczynku zawieszenia wzdłuż odległości zawieszenia (wartość domyślna to 0,5, co odpowiada zachowaniu zwykłego zawieszenia samochodu)
- Forward/Sideways Friction - właściwości tarcia opony podczas toczenia się koła do przodu i na boki

5.1.5. Prefabs

System prefabów, ukazany w dokumentacji [10], umożliwia tworzenie, konfigurowanie i przechowywanie GameObject wraz ze wszystkimi jego komponentami, wartościami właściwości i potomnymi GameObjects jako zasób wielokrotnego użytku. Taki zasób działa jak szablon, na podstawie którego można tworzyć nowe wystąpienia prefabów w scenie. Wszelkie zmiany wprowadzane do zasobu są automatycznie odzwierciedlane w instancjach tego prefabu, co pozwala łatwo tworzyć szerokie zmiany w całym projekcie bez konieczności wielokrotnego wprowadzania tej samej edycji w każdej kopii zasobu. Można zagnieżdżać prefaby w innych prefabrykatach, aby tworzyć złożone hierarchie obiektów, które można łatwo edytować na wielu poziomach. Nie oznacza to jednak, że wszystkie instancje prefabów muszą być identyczne, ponieważ można zastąpić ustawienia poszczególnych instancji prefabów, aby niektóre instancje prefabów różniły się od innych. Istnieje też opcja tworzenia wariantów prefabrykatów, które pozwalają grupować zestaw przesłonięć w znaczącą odmianę prefabrykatu. W celu utworzenia

prefabu należy stworzyć nowy pusty prefab poprzez wykorzystanie listy rozwijanej w widoku Project Create -> Prefab bądź wybrać z menu górnego Assets -> Create -> Prefab. Nowy prefab to pusty pojemnik (szara ikonka), który czeka na zapełnienie go GameObjectami (ikonka zmienia kolor na niebieski). Wypełnienie prostego prefabu polega na przeciągnięciu GameObject, który znajduje się na scenie do pustego prefabu. W tym momencie nazwa prefabu zmienia kolor na niebieski, aby zaznaczyć, że orginalny GameObject z widoku Hierarchy stał się instancją prefabu. Innym sposobem jest wypełnienie pustego prefabu GameObjectem z widoku Projektu i ustawienie jego instancji na scenie.

5.2. Zapisywanie, wczytywanie obiektów

Wczytywanie prefabów zapisanych poprzez przycisk Save Robot lub nowo dodanych ze sceny Menu odbywa się poprzez metodę Resources.LoadAll zamieszczoną w kodzie 5.1. Analogiczne kroki wykonuje się dla środowiska.

Listing 5.1: Funkcja dodaje wszystkie roboty do list z folderu Assets/Resources/Robots/ : `SearchForRobots`.

```
public static void SearchForRobots() {
 robotsFound.Clear();
 GameObject[] robots = Resources.LoadAll < GameObject > ("Robots/");
 foreach(GameObject r in robots) {
  if (r.GetComponent < Robot > ()) {
   robotsFound.Add(r);
  }
 }
}
```

Za okno dialogowe wyboru pliku odpowiada wbudowana w Unity fukcja EditorUtility.OpenFilePanel, która pozwala ograniczyć wybór plików o niewłaściwym rozszerzeniu. Użycie tej metody zostało pokazne w kodzie 5.2.

Listing 5.2: Metoda otwiera okno dialogowe wyboru pliku .fbx i zwraca ścieżkę:`EditorUtility.OpenFilePanel`.

```
string path = EditorUtility.OpenFilePanel("Select␣model␣file...", "", "FBX");
```

5.3. Statusy w symulacji

Symulacja posiada statusy pokazane w kodzie 5.3, które odpowiadają aktualnym działaniom symulacji. W scenie Menu, status ustawiany jest na menu, a po przejściu do następnej sceny statusem jest starting, podczas którego sprawdzane jest czy wszystkie roboty posiadają silnik oraz program. Gdy wykryty zostanie brak silnika, status zmieniany jest na edit. Po zakończeniu edycji robota, status ulega modyfikacji na starting celem sprawdzenia robotów. Jeżeli wszystko działa poprawnie, status przeobrażany jest na stopped, z którego poprzez przycisk Start, przechodzi się do statusu simulating. Przyciskając przycisk Stop, status zmieniany jest na stopped, z którego przy pomocy przycisku Edit można przejść do statusu edit. Zamykając aplikację statusowi przypisywany jest parametr end.

Listing 5.3: Statusy w symulacji: `State`.

```
public enum State {
 menu,
 starting,
 simulating,
 stopped,
 edit,
 end
}
```

5.4. Robot program

Każdy program robota jest uruchamiany jako nowy wątek, zgodnie z kodem 5.4, dzięki czemu nie wpływa on na symulację. To rozwiązanie umożliwia korzystanie z takich funkcji jak Thread.Sleep, która zatrzymuje wątek. Kod robota musi być napisany w pętli, aby mógł się cały czas wykonywać.

Listing 5.4: Tworzenie nowego wątku z programem robota: `StartRobot`.

```
public void StartRobot() {
 if (methods != null) {
  program = new Thread(() => {
   methods[0].Invoke(this, new object[] {
    this
   });
  });
  StartSensor();
  program.Start();
 } else
  Debug.Log("Robot␣" + name + "␣hasn't␣got␣program!");
}
```

5.5. Kompilator

Do interpretowania kodu wprowadzonego przez użytkownika została wykorzystana biblioteka System.CodeDom.Compiler, opisana w dokumentacji [1], zastosowana we fragmencie kodu 5.5. Pozwala ona skompilować tekst na język maszynowy przy zachowaniu podanych parametrów. Do programu dołączane są biblioteki systemowe, MonoBehaviour do obsługi metod Unity oraz skrypt Robot, który umożliwia korzystanie ze zmiennych sensora i silnika.

Listing 5.5: Fragment kodu odpowiadającego za ustawienie parametrów kompilatora: `Compilator`.

```
ICodeCompiler icc = codeProvider.CreateCompiler();
string Output = robot.nameWithoutSpace + ".dll";
string OutputCSCode = robot.code + ".txt";
textError.text = "";
System.CodeDom.Compiler.CompilerParameters parameters = new CompilerParameters();
parameters.GenerateExecutable = false;
parameters.GenerateInMemory = true;
parameters.ReferencedAssemblies.Add("System.dll");
parameters.ReferencedAssemblies.Add(typeof(Simulation).Assembly.Location);
parameters.ReferencedAssemblies.Add(typeof(Robot).Assembly.Location);
parameters.ReferencedAssemblies.Add(typeof(MonoBehaviour).Assembly.Location);
```

```
parameters.OutputAssembly = Output;
System.IO.File.WriteAllText(OutputCSCode, textCode.text);
string code = RenameVariables(textCode.text, robot);
CompilerResults results = icc.CompileAssemblyFromSource(parameters, code);
```

Biblioteka System.CodeDom.Compiler zwraca także informacje o powodzeniu bądź niepowodzeniu kompilacji tak jak zostało pokazane w kodzie 5.6. W przypadku, gdy otrzymania błędu, podaje ona pełną informację o błędzie oraz której linii dotyczy ten błąd. W momencie udanej kompilacji zwracany jest odpowiedni komunikat, a program ładowany jest do robota. Zmieniona zostaje również treść string nameWithoutSpace, która jest także nazwą utworzonego przez kompilator pliku .dll. Niezbędne jest to, aby za każdym razem do robota dołączać nowy plik .dll, wcześniej nie dodawany w symulacji. Jest to spowodowane brakiem możliwości odłączenia pliku .dll od aplikacji, co dokładnie wyjaśnia w swoim artykule [22] Jason Zander.

Listing 5.6: Fragment kodu odpowiadającego za zwrócenie informacji o kompilacji oraz zmiany nazwy pliku .dll: `Compilator`.

```
if (results.Errors.Count > 0) {
 textError.faceColor = Color.red;
 foreach(CompilerError CompErr in results.Errors) {
  textError.text = textError.text +
   "Line␣number␣" + CompErr.Line +
   ",␣Error␣Number:␣" + CompErr.ErrorNumber +
   ",␣'" + CompErr.ErrorText + ";" +
   "\n" + "␣\n";
 }
} else {
 robot.initializationCode();
 int count = 1;
 string tempFileName = robot.nameWithoutSpace;
 patchsToDelete.Add(Output);
 while (File.Exists(Output)) {
  tempFileName = string.Format("{0}_{1}", robot.code, count++);
  Output = tempFileName + ".dll";
 }
 robot.nameWithoutSpace = tempFileName;
 textError.faceColor = Color.blue;
 textError.text = "Success!";
}
```

Przed skompilowaniem programu robota zostają zamienione zmienne o nazwach silnika i sensora, na zmienne interpretowane przez symulator. Modyfikacji ulega także funkcja Main() na Main(Robot), co pozwala na wysłanie referencji robota do programu. Za te zmiany odpowiedzialna jest funkcja RenameVariables zaprezentowana w kodzie 5.7.

Listing 5.7: Funkcja zmieniająca podane zmienne w kodzie na interpretowane dane: `RenameVariables`.

```
private string RenameVariables(string text, Robot robot) {
 if (!robot) return text;
 text = text.Replace("Code", robot.nameWithoutSpace);
 text = text.Replace("void␣Main()", "void␣Main(Robot␣robot)");
 for (int i = 0; i < robot.motors.Length; i++)
  text = text.Replace(Simulation.robotSelected.motors[i].name,
                      "robot.motors[" + i + "].powerMotor");
 for (int i = 0; i < robot.sensors.Length; i++)
  text = text.Replace(Simulation.robotSelected.sensors[i].name,
                      "robot.sensors[" + i + "].detected");
 return text;
}
```

5.6. Silnik

Silnikiem sterujemy przy pomocy zmiennej powerMotor pokazanej w kodzie 5.8, która odpowiednio przy dodatniej wartości powoduje kręcenie koła do przodu, analogicznie przy ujemnej wartości kręci kołem do tyłu. Kolejnym krokiem jest wyznaczenie Torque, czyli momentu obrotowego ograniczonego przez maksymalny moment obrotowy silnika. Za ograniczenie to odpowiada funkcja Mathf.Clamp, a zwracana wartość poddawana jest komponentowi wheelCollider przez zmienną motorTorque.

Listing 5.8: Funkcja poruszania się koła: Run.

```
public void Run() {
 torque = (powerMotor * maxWheelRpm - wheelCollider.rpm) * Kp;
 wheelCollider.motorTorque = Mathf.Clamp(torque, -maxWheelTorque, maxWheelTorque);
}
```

5.7. Czujnik odległościowy

Symulowanie czujnika odległościowego pokazanego w kodzie 5.9, odbywa się poprzez wyznaczanie linii zakreślających pole widzenia FOV, przy ograniczonym zasięgu maxRange. Dla każdej linii ustalany jest punkt direction, który wyznacza kierunek.

Listing 5.9: Funkcja skanująca obszar widzenia sensora: UpdateData.

```
private IEnumerator UpdateData() {
 while (scanning) {
  float proximity = maxRange;
  for (float a = -FOV; a < FOV; a += 2 f) {
   Quaternion rotation = Quaternion.Euler(new Vector3(0, a, 0));
   Vector3 direction = rotation * startLight.forward;
   float check = Cast(direction);
   if (check < proximity) proximity = check;
  }
  direction = startLight.forward * proximity;
  if (proximity < maxRange * 0.75 f) detected = true;
  else detected = false;
  yield
  return new WaitForSeconds(updateDt);
 }
}
```

Rozdział 6

Podsumowanie

W omówionej pracy inżynierskiej zrealizowano założony cel, jakim było stworzenie symulatora robotów mobilnych umożliwiającego użytkownikowi sprawdzenie ich algorytmu na własnym trójwymiarowym modelu robota na samodzielnie zaprojektowanym środowisku. Fakt, że symulację oprogramowano w języku C# zapewnił silne tworzenie kopii zapasowych pamięci. Język programowania C# zawiera kopię zapasową wysokiej pamięci, dzięki czemu problem wycieku pamięci nie występuje, jak to ma miejsce w przypadku języka C ++. Język C# jest łatwy w rozwoju, ponieważ ma bogatą klasę bibliotek, które ułatwiają implementację wielu funkcji. Innym wartym uwagi rozwiązaniem w C# jest odśmiecacz, tak zwany garbage collector, zajmujący się automatycznym zwalnianiem pamięci po obiektach, które nie będą już wykorzystywane. Co więcej kod C# podczas kompilacji generuje plik „.exe" lub „.dll", który jest również nazywany przenośnym plikiem wykonywalnym, zawierającym kod Microsoft Intermediate Language (MSIL).

Unity to potężny silnik, który ma wiele wbudowanych komponentów fizyki. Obsługuje on symulacje fizyczne, a dostosowując kilka ustawień parametrów, można stworzyć obiekt, który zachowuje się w realistyczny sposób. Kontrola fizyki za pomocą skryptów nadaje obiektowi dynamikę pojazdu, maszyny, materiału, a wbudowane komponenty są bardzo przydatne w szybkim rozwoju. Ważnym elementem jest Rigidbody, który umożliwia fizyczne zachowanie obiektu, gdzie obiekt do którego przymocowane jest ciało sztywne może reagować na grawitację. Ponadto kluczową rolę odgrywa Collider jako jeden z najważniejszych wbudowanych komponentów Unity3D, służący do definiowania kształtu zderzenia fizycznego. Unity umożliwia wygenerowanie projektu do aplikacji, ale jest to niemożliwe w tym projekcie ze względu na brak ustawiania pozycji obiektów oraz kompilacje programu do robota, które zapewnia edytor Unity.

Debugowanie i poprawianie jest łatwiejsze w Unity, ponieważ podczas symulacji wyświetlane są wszystkie jej zmienne, co z kolei umożliwia programistom debugowanie procesu w czasie wykonywania. Tryb odtwarzania, który służy do szybkiej edycji iteracyjnej, pozwala w dowolnym momencie, jeśli napotka się błąd lub jakaś funkcja nie działa zgodnie z oczekiwaniami, wstrzymać jej działanie i zmienić kod zgodnie z własnymi upodobaniami. Aby jeszcze łatwiej debugować, istnieje możliwość przeglądania symulacji klatka po klatce i wskazywania problemów.

Zaprezentowaną symulację można rozbudować o kolejne czujniki np. akcelerometr, enkoder lub kompas, co pozwoli na kontrolowanie przemieszczania się robota w programie. Zastosowanie żyroskopu umożliwiłoby symulowanie robotów balansujących, a czujniki odbiciowe usprawniłyby symulowanie robotów typu sumo czy LineFolower. Diody i przyciski pozwoliłyby na sygnalizowanie lub zmienianie stanu robota - start,stop. Kontroler baterii ograniczyłby czas działania robota, a wprowadzenie kamer umożliwiłoby testowanie trudnych algorytmów rozpoznawania robota, które nie są w stanie być używane w rzeczywistych robotach ze względu na słaby sprzęt. Wszystkie te usprawnienia ułatwia wykorzystanie Unity.

Literatura

[1] C# system.codedom.compiler. Web pages: https://docs.microsoft.com/pl-pl/dotnet/api/system.codedom.compiler?view=netframework-4.8 [dostęp dnia 20 listopada 2019].

[2] Camera. Web pages: https://docs.unity3d.com/Manual/class-Camera.html [dostęp dnia 3 grudnia 2019].

[3] Colliders. Web pages: https://docs.unity3d.com/Manual/CollidersOverview.html [dostęp dnia 25 listopada 2019].

[4] Gameobjects. Web pages: https://docs.unity3d.com/Manual/GameObjects.html [dostęp dnia 29 listopada 2019].

[5] Gameview. Web pages: https://docs.unity3d.com/Manual/GameView.html [dostęp dnia 3 grudnia 2019].

[6] Git – co, gdzie, jak i dlaczego? Web pages: https://www.nettecode.com/git-dlaczego-materialy-start/ [dostęp dnia 28 listopada 2019].

[7] Github – podstawy. Web pages: http://devfoundry.pl/github-podstawy/ [dostęp dnia 28 listopada 2019].

[8] Hierarchy. Web pages: https://docs.unity3d.com/Manual/Hierarchy.html [dostęp dnia 3 grudnia 2019].

[9] Lighting. Web pages: https://docs.unity3d.com/Manual/Lighting.html [dostęp dnia 3 grudnia 2019].

[10] Prefabs. Web pages: https://docs.unity3d.com/Manual/Prefabs.html [dostęp dnia 25 listopada 2019].

[11] Rigidbody. Web pages: https://docs.unity3d.com/Manual/class-Rigidbody.html [dostęp dnia 25 listopada 2019].

[12] Scenes. Web pages: https://docs.unity3d.com/Manual/CreatingScenes.html.

[13] Szybki start: pierwsze spojrzenie na środowisko ide programu visual studio. Web pages: https://docs.microsoft.com/pl-pl/visualstudio/ide/quickstart-ide-orientation?view=vs-2019 [dostęp dnia 28 listopada 2019].

[14] Teoria robotyki. Web pages: https://www.robotyka.com/teoria.php/teoria.5 [dostęp dnia 12 listopada 2019].

[15] Unity opis bibliotek. Web pages: https://docs.unity3d.com/Manual/index.html [dostęp dnia 25 listopada 2019].

[16] Usingtheinspector. Web pages: https://docs.unity3d.com/Manual/UsingTheInspector.html [dostęp dnia 3 grudnia 2019].

[17] Wheelcollider. Web pages: https://docs.unity3d.com/Manual/class-WheelCollider.html [dostęp dnia 29 listopada 2019].

[18] J. Hocking. *Unity w akcji*. Helion, 2017.

[19] K. JAGACIAK. C# w unity. poznaj narzędzie do tworzenia gier. Web pages: https://www.komputerswiat.pl/poradniki/programy/c-w-unity-poznaj-narzedzie-do-tworzenia-gier/7yx77qg [dostęp dnia 12 października 2019].

[20] W. Mariusz J. Giergiel, Zennon Hendzel. *Modelowanie i sterowanie mobilnych robotów kołowych*. Wydawnictwo Naukowe PWN, 2013.

[21] K. Trybulec. Metody wirtualne, abstrakcyjne i polimorfizm w c#. Web pages: https://www.p-programowanie.pl/c-sharp/metody-wirtualne-abstrakcyjne-i-polimorfizm/ [dostęp dnia 22 października 2019].

[22] J. Zander. Why isn't there an assembly.unload method? Web pages: https://blogs.msdn.microsoft.com/jasonz/2004/05/31/why-isnt-there-an-assembly-unload-method/ [dostęp dnia 20 listopada 2019].

Dodatek A

Opis załączonej płyty CD

Na załączonej płycie cd znajduje się projekt pracy inżynierskiej do uruchomienia w środowisku Unity w wersji 2019.2.5f1.