Symulator robota koszącego

Dokumentacja końcowa,
19 stycznia 2026

Zrealizowane funkcjonalności

- Implementacja prostokątnego trawnika, jako ciągłej przestrzeni.
- Użytkownik może skorzystać z metod kosiarki i własnych wskaźników, aby samodzielnie określić obszar do wykoszenia
- Podstawowe funkcje robota: włączanie/wyłączanie mechanizmu koszenia, obrót o dowolny kąt z dokładnością do 1 stopnia, przejazd o zadaną odległość.
- Wykrywanie próby opuszczenia wyznaczonego obszaru i sygnalizowanie tego zdarzenia do programu sterującego.
- Interfejs graficzny (GUI) przedstawiający w czasie rzeczywistym proces koszenia.
- Możliwość przyspieszenia upływu czasu symulacji.
- Zapisywanie szczegółowych logów z akcji podejmowanych przez robota do pliku tekstowego.

Nie realizowaliśmy funkcjonalności, oznaczonych w dokumentacji wstępnej jako *opcjonalne*:
- Wprowadzenie do symulatora efektów dźwiękowych, np.odgłosów pracy kosiarki.
- Gromadzenie i prezentowanie statystyk dotyczących procesu koszenia

Napotkane problemy

- Głównym napotkanym problemem przy implementacji funkcji koszącej była bardzo niska wydajność koszenia, spowodowana błędną koncepcją koszenia. Pierwotnie koszenie miało odbywać się na zasadzie wycinania okrągłych obszarów w pętli przechodząc po kolejnych polach. Jednak powodowało to wielokrotnym wykaszanie tych samych pól, co nie miało sensu. Postanowiliśmy zmienić technikę koszenia na koszenie dwóch kół na końcach koszonego obszaru oraz prostokąta, w którym pola koszone były tylko raz. Dzięki temu w bardzo dużym stopniu poprawić wydajność.

- Innym problemem było zaimplementowanie rozdziału komend zajmujących więcej czasu w symulacji, która podzielona jest na poszczególne kroki co 20ms. Niezbędne było utworzenie logiki, która ‘rozbija’ treść komendy i pozwala głównemu silnikowi symulacji na powrót i realizację zadanej komendy w poszczególnych krokach symulacji.

- Problemem było też to, że nie chcieliśmy zmuszać użytkownika do długiego pobierania i samodzielnego kompilowania biblioteki Qt ze źródeł. W skrypcie użyte są więc na gotowe pakiety Ubuntu, które niestety nie pobierają Qt lokalnie. Przez to nasz program nie jest też przenośny na Windows.

Opis architektury aplikacji

Poniżej załączona jest analiza lini kodu w projekcie.
73 text files, 71 unique files, 12 files ignored.

Language	files	blank	comment	code
C++	39	1472	277	4851
C/C++ Header	28	179	156	620
HTML	1	63	0	451
Markdown	1	35	0	128
Bourne Shell	1	8	0	56
CMake	1	23	4	56
SUM:	**71**	**1780**	**437**	**6162**

Podstawowe klasy

Mower i *Lawn* reprezentują kosiarkę oraz obszar pracy z trawą, a *Point* służy do oznaczania celów na mapie. Stałe prędkości i ustawienia techniczne systemu zapisane są w pliku *Constants* oraz *Config*.

Przetwarzanie kodu od użytkownika

Zaangażowane klasy: *MowerController* → *Commands* → *Engine* → *StateSimulation*
Użytkownik poprzez *MowerController* dodaje zadania *Command* do kolejki działań. Koordynujący pracę *Engine* w każdym kroku symulacji wywołuje wykonanie bieżącej komendy z tej kolejki. *Command* modyfikuje stan świata wyłącznie poprzez metody klasy *StateSimulation*, która odpowiada za przebieg i fizykę symulacji.

Symulacja i wizualizacja

Zaangażowane klasy: *StateSimulation* → *Engine* → *StateInterpolator i RenderTimeController* →*Visualizer*
Na koniec każdego kroku symulacji *Engine* pobiera aktualny stan z klasy *StateSimulation* w formie migawki *SimulationSnapshot*, który przekazywany jest do klasy *StateInterpolator*. *StateInterpolator* przechowuje bufor takich migawek i na podstawie informacji o aktualnym czasie w wizualizacji (który wyliczany jest przez *RenderTimeController*) upewnia się, że zmiana kąta i pozycji kosiarki następuje płynnie. Tak przygotowane dane są pobierane przez klasę *Visualizer*, która za pomocą biblioteki *Qt* rysuje w odpowiedniej skali aktualny stan trawnika, punktów i kosiarki.

Logowanie zdarzeń i ruchów kosiarki

Zaangażowane klasy: Log ← Logger← State simulation → Engine → FileLogger
W trakcie symulacji klasa *StateSimulation* generuje obiekty *Log* dokumentujące zdarzenia oraz błędy i przekazuje je do kolejki w klasie *Logger*. Następnie *Engine* w każdym kroku pobiera zgromadzone wpisy i wykorzystuje *FileLogger* do ich trwałego zapisu na dysku.

Testy

W naszym projekcie, w folderze *tests/* mamy 13 plików zawierających różnego rodzaju testy jednostkowe. Łącznie jest 212 testów, które **pokrywają 80.7% linii kodu, 88.9% funkcji oraz 70.2% gałęzi.**.

Dokumentacja wygenerowana z doxyfile

Aby wygenerować dokumentację należy skorzystać z Doxygen i Graphviz, które instalowane są w ramach skryptu setup_extra_libraries. Następnie należy wykonać następujące kroki:

- doxygen Doxyfile
- xdg-open html/index.html

Pracochłonność zrealizowanych zadań

Zadanie	**Szacowany czas wykonania** (w godzinach)	**Rzeczywisty czas pracy**
Utworzenie struktury katalogów w projekcie	2	2
Automatyzacja kompilacji za pomocą CMake	3	3
Plik konfiguracyjny zawierający stałe programowe	2	5
Stworzenie klas bazowych (np. *Lawn*, *LawnField*, *Mower*)	5	6
Implementacja mechaniki ruchu i obrotu kosiarki	5	10
Orientacja kosiarki względem położenia znaczników	5	4
Detekcja próby wyjścia za granice środowiska	3	3
Powstrzymanie wyjścia poza granice środowiska	3	3
Implementacja procesu koszenia	6	11 (zmiana koncepcji)
Implementacja wyznaczania znaczników	5	6
Szczegółowe zaprojektowanie wielowątkowości	6	5
Wprowadzenie usypiania wątku symulacji	5	5
Wprowadzenie komunikacji pomiędzy wątkami	6	7
Zaprojektowanie oprawy graficznej symulacji	6	6
Wizualizacja stanu trawnika	6	10
Wizualizacja położenia kosiarki	5	6
Interpolacja wizualizacji		10
Wizualizacja znaczników użytkownika	4	4
Przyspieszenie czasu wykonania symulacji	3	6
Implementacja systemu logowania działań kosiarki	2	3
Suma godzin:	**101**	**115**