DOKUMENTACJA

Aircaft Carrier Problem

Paweł Buczek (173599) L01 EF-DI
Bartłomiej Krówka (173650) L01 EF-DI

1. Zadanie (nr. 9)

On the aircraft carrier, planes touch down and soar into the skies. To do this, they require sole rights to the runway. The carrier has room for 10 planes. If there are fewer than 5 planes, landing aircraft get first dibs on the runway; otherwise, it's takeoff planes that get the priority. If the number of planes surpasses 10, some of them remain airborne, patiently awaiting their turn.

2. Cel projektu

Celem projektu jest symulowanie zarządzania lądowaniem i startowaniem samolotów na lotniskowcu. Program powinien umożliwić priorytetowe lądowanie samolotów, gdy ich liczba jest mniejsza niż 5, oraz priorytetowy start, gdy liczba samolotów wynosi 5 lub więcej. Dodatkowo, system powinien uwzględniać sytuację, gdy liczba samolotów jest większa niż liczba wolnych miejsc na lotniskowcu.

3. Rozwiązanie problemu

Lotniskowiec ma 10 miejsc. Samoloty lądujące i startujące są kolejkowane. Za obsługę kolejek oraz zarządzanie dostępem do pasa odpowiadają trzy różne wątki. Gdy samolotów znajdujących się na lotniskowcu jest mniej niż 5 oraz kolejka samolotów, które chcą lądować jest pusta to pas jest udostępniany dla samolotów startujących. W przypadku, kiedy na lotniskowcu jest 5 lub więcej samolotów, a kolejka samolotów chcących startować jest pusta, to pas jest udostępniany dla samolotów lądujących aż do momentu kiedy nie skończą się wolne miejsca.

4. Funkcjonalność

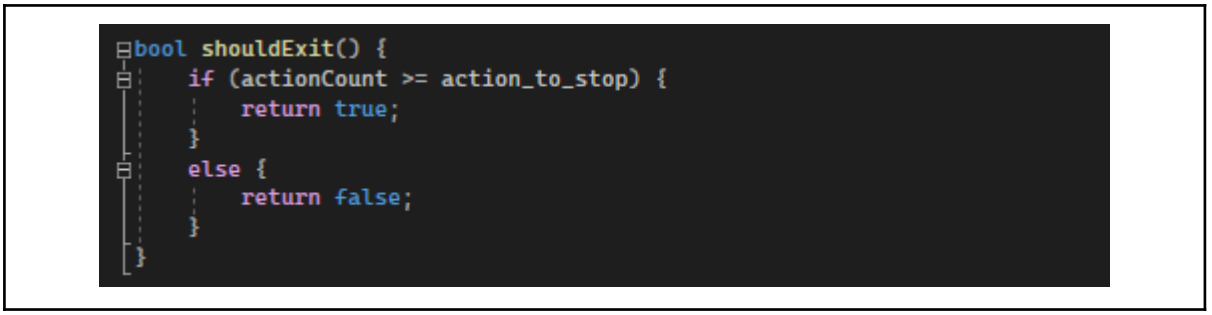
Program symuluje zarządzanie lądowaniem i startowaniem samolotów w zależności od ich aktualnej liczby na lotniskowcu. Umożliwia użytkownikowi sterowanie czasem wykonywania poszczególnych wątków oraz ilością dozwolonych akcji programu. Każda wykonywania akcja jest raportowana w konsoli. Dodatkowo program wizualizuje graficznie aktualny stan lotniskowca.

5. Technologie

- język programowania: C++
- biblioteki: iostream, string, thread, chrono, mutex, vector, algorithm, random, queue
- GUI: SFML
- IDE: Microsoft Visual Studio 2022 Community Edition

6. Opis funkcji:

`Funkcja shouldExit`



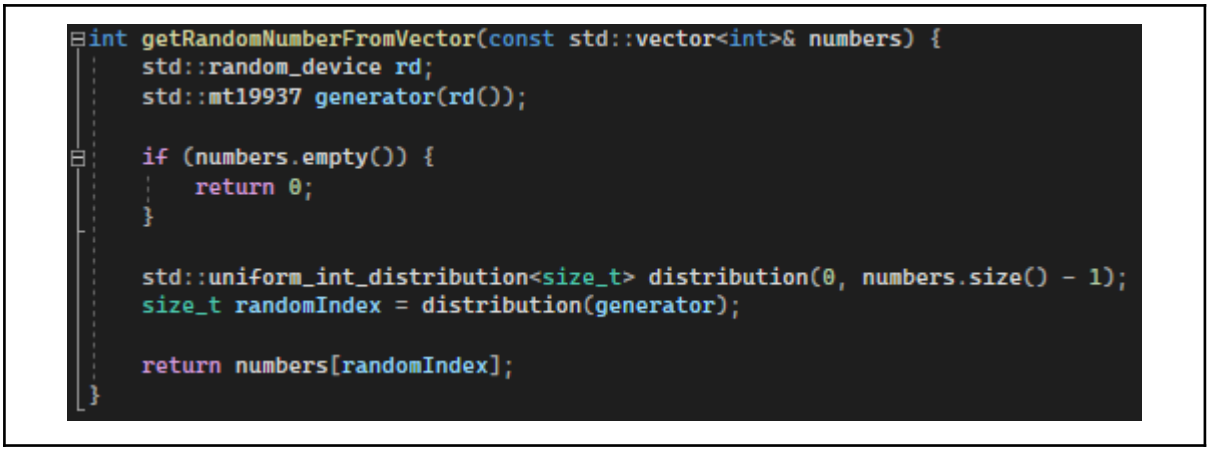
```
bool shouldExit() {
    if (actionCount >= action_to_stop) {
        return true;
    }
    else {
        return false;
    }
}
```

Funkcja używana do kontrolowania, kiedy program powinien się zakończyć.

Działanie funkcji:

- jeśli liczba wykonanych akcji przekroczy określoną wartość zwraca `true`,
- w przeciwnym wypadku zwraca `false`

Funkcja getRandomNumberFromVector



```cpp
int getRandomNumberFromVector(const std::vector<int>& numbers) {
    std::random_device rd;
    std::mt19937 generator(rd());

    if (numbers.empty()) {
        return 0;
    }

    std::uniform_int_distribution<size_t> distribution(0, numbers.size() - 1);
    size_t randomIndex = distribution(generator);

    return numbers[randomIndex];
}
```

Funkcja zwracająca losową liczbę z podanego jako argument wektora liczb.

Działanie funkcji:

1. **Inicjalizacja generatora liczb pseudolosowych:**
   Funkcja rozpoczyna od inicjalizacji generatora liczb pseudolosowych za pomocą `std::random_device` i `std::mt19937`.

2. **Sprawdzenie, czy wektor jest pusty:**
   Następnie funkcja sprawdza, czy wektor numbers jest pusty. Jeśli tak, funkcja zwraca 0 jako wartość domyślną.

3. **Utworzenie rozkładu jednorodnego:**
   Jeśli wektor nie jest pusty, funkcja tworzy rozkład jednorodny dla zakresu indeksów wektora za pomocą `std::uniform_int_distribution`. Rozkład jednorodny to rodzaj rozkładu prawdopodobieństwa, w którym każda liczba ma równą szansę na bycie wybraną.

4. **Wybór losowego indeksu:**
   Funkcja następnie wybiera losowy indeks z tego rozkładu. Indeks ten jest wybierany za pomocą wcześniej zainicjowanego generatora liczb pseudolosowych.

5. **Zwrócenie liczby z wektora o wylosowanym indeksie:**
   Na koniec funkcja zwraca liczbę z wektora numbers o wylosowanym indeksie.

Funkcja landing

```
void landing() {
    srand(static_cast<unsigned>(time(nullptr)));
    while (!shouldExit()) {
        int plane = -1;

        vector<int> notAircraftCarrier;

        notAircraftCarrier.clear();

        for (int i = 1; i <= 20; ++i) {
            if (std::find(aircraftCarrier.begin(), aircraftCarrier.end(), i) == aircraftCarrier.end()) {
                notAircraftCarrier.push_back(i);
            }
        }

        int number = getRandomNumberFromVector(notAircraftCarrier);

        {
            lock_guard<mutex> lock(aircraftMutex);
            queue<int> copyLanding = landingQueue;
            bool inQueue = false;
            while (!copyLanding.empty()) {
                if (copyLanding.front() == number) {
                    inQueue = true;
                    break;
                }
                copyLanding.pop();
            }

            if (aircraftCarrier.size() > 0) {
                if (find(aircraftCarrier.begin(), aircraftCarrier.end(), number) == aircraftCarrier.end() && !inQueue) {
                    plane = number;
                }
            }
            else {
                if (!inQueue) {
                    plane = number;
                }
            }

            if (plane != -1) {
                landingQueue.push(plane);
                cout << "Plane " << plane << " added to landing queue." << endl;
                cout << "Current planes on landing queue: " << landingQueue.size() << " [ ";
                copyLanding = landingQueue;
                while (!copyLanding.empty()) {
                    cout << copyLanding.front() << " ";
                    copyLanding.pop();
                }
                cout << "]" << endl;
            }
        }
        this_thread::sleep_for(chrono::milliseconds(1500 + rand() % landingTime));
    }
}
```

Funkcja odpowiedzialna za symulowanie procesu lądowania samolotów na lotniskowcu.

Działanie funkcji:

1. **Inicjalizacja generatora liczb pseudolosowych:**
   Funkcja rozpoczyna działanie od inicjalizacji generatora liczb pseudolosowych za pomocą `srand`.

2. **Wejście w pętlę:**
   Funkcja wchodzi w pętlę, która kontynuuje działanie, dopóki funkcja `shouldExit()` nie zwróci `true`.

3. **Inicjalizacja wektora:**
   Wypełnienie wektora `notAircraftCarrier` numerami samolotów, które nie są obecnie na lotniskowcu.

4. **Losowanie numeru samolotu:**
   Losowanie numeru samolotu z wektora `notAircraftCarrier` za pomocą funkcji `getRandomNumberFromVector`.

5. **Sprawdzanie, czy samolot jest już w kolejce do lądowania:**
   Sprawdzenie, czy wylosowany samolot jest już w kolejce do lądowania. Jeśli tak, samolot nie jest dodawany do kolejki.

6. **Dodawanie samolotu do kolejki do lądowania:**
   Jeśli samolot nie jest jeszcze w kolejce do lądowania, jest do niej dodawany.

7. **Wyświetlanie informacji o kolejce do lądowania:**
   Wyświetlenie informacji o aktualnym stanie kolejki do lądowania w konsoli.

8. **Opóźnienie:**
   Na koniec każdej iteracji pętli funkcja wprowadza losowe opóźnienie, które jest zależne od zmiennej `landingTime`, ale nie krótsze niż 1500 milisekund.

Funkcja starting

```
void starting() {
    srand(static_cast<unsigned>(time(nullptr)));
    while (!shouldExit()) {
        int positionPlane = -1;
        queue<int> copyStarting;
        {
            lock_guard<mutex> lock(aircraftMutex);
            if (aircraftCarrier.size() > 0) {
                int plane = getRandomNumberFromVector(aircraftCarrier);
                auto i = find(aircraftCarrier.begin(), aircraftCarrier.end(), plane);

                copyStarting = startingQueue;
                bool inQueue = false;
                while (!copyStarting.empty()) {
                    if (copyStarting.front() == plane) {
                        inQueue = true;
                        break;
                    }
                    copyStarting.pop();
                }

                if (i != aircraftCarrier.end() && !inQueue) {
                    positionPlane = distance(aircraftCarrier.begin(), i);
                }
            }

            if (positionPlane != -1) {
                startingQueue.push(aircraftCarrier[positionPlane]);
                cout << "Plane " << aircraftCarrier[positionPlane] << " added to starting queue." << endl;
                cout << "Current planes on starting queue: " << startingQueue.size() << " [ ";
                copyStarting = startingQueue;
                while (!copyStarting.empty()) {
                    cout << copyStarting.front() << " ";
                    copyStarting.pop();
                }
                cout << "]" << endl;
            }
        }
        this_thread::sleep_for(chrono::milliseconds(1500 + rand() % startingTime));
    }
}
```

Funkcja odpowiedzialna za symulowanie procesu startowania samolotów na lotniskowcu.

Działanie funkcji:

1. **Inicjalizacja generatora liczb pseudolosowych:**
   Funkcja rozpoczyna od inicjalizacji generatora liczb pseudolosowych za pomocą `srand`.

2. **Wejście w pętlę:**
   Funkcja wchodzi w pętlę, która kontynuuje działanie, dopóki funkcja `shouldExit()` nie zwróci `true`.

3. **Sprawdzanie, czy na lotniskowcu są samoloty:**
   Sprawdzenie, czy na lotniskowcu (`aircraftCarrier`) są jakiekolwiek samoloty. Jeśli tak, to wybiera losowy samolot z lotniskowca za pomocą funkcji `getRandomNumberFromVector`.

4. **Sprawdzanie, czy samolot jest już w kolejce do startu:**
   Sprawdzenie, czy wybrany samolot jest już w kolejce do startu. Jeśli tak, samolot nie jest dodawany do kolejki.

5. **Dodawanie samolotu do kolejki do startu:**
   Jeśli samolot nie jest jeszcze w kolejce do startu, jest do niej dodawany.

6. **Wyświetlanie informacji o kolejce do startu:**
   Wyświetlenie informacji o aktualnym stanie kolejki do startu w konsoli.

7. **Opóźnienie:**
   Na koniec każdej iteracji pętli funkcja wprowadza losowe opóźnienie, które jest zależne od zmiennej `startingTime`, ale nie krótsze niż 1500 milisekund.

Funkcja action

```
void action() {
    srand(static_cast<unsigned>(time(nullptr)));
    while (!shouldExit()) {
        {
            lock_guard<mutex> lock(aircraftMutex);
            if (aircraftCarrier.size() < borderlineNumber) {
                if (!landingQueue.empty()) {
                    int plane = landingQueue.front();
                    if (aircraftCarrier.size() < 10) {
                        landingQueue.pop();
                        aircraftCarrier.push_back(plane);
                        cout << "Plane " << plane << " is landing..." << endl;
                        cout << "\nCurrent planes on carrier: " << aircraftCarrier.size() << " [ ";
                        for (size_t i = 0; i < aircraftCarrier.size(); ++i) {
                            cout << aircraftCarrier[i] << " ";
                        }
                        cout << "]\n" << endl;
                        actionCount++;
                    }
                }
                else if (!startingQueue.empty()) {
                    int planeNumber = startingQueue.front();
                    auto it = find(aircraftCarrier.begin(), aircraftCarrier.end(), planeNumber);
                    if (it != aircraftCarrier.end()) {
                        startingQueue.pop();
                        cout << "Plane " << *it << " is starting..." << endl;
                        aircraftCarrier.erase(it);
                        cout << "\nCurrent planes on carrier: " << aircraftCarrier.size() << " [ ";
                        for (size_t i = 0; i < aircraftCarrier.size(); ++i) {
                            cout << aircraftCarrier[i] << " ";
                        }
                        cout << "]\n" << endl;
                        actionCount++;
                    }
                }
            }
            else {
                if (!startingQueue.empty()) {
                    int planeNumber = startingQueue.front();
                    auto it = find(aircraftCarrier.begin(), aircraftCarrier.end(), planeNumber);
                    if (it != aircraftCarrier.end()) {
                        startingQueue.pop();
                        cout << "Plane " << *it << " is starting..." << endl;
                        aircraftCarrier.erase(it);
                        cout << "\nCurrent planes on carrier: " << aircraftCarrier.size() << " [ ";
                        for (size_t i = 0; i < aircraftCarrier.size(); ++i) {
                            cout << aircraftCarrier[i] << " ";
                        }
                        cout << "]\n" << endl;
                        actionCount++;
                    }
                }
                else if (!landingQueue.empty()) {
                    int plane = landingQueue.front();
                    if (aircraftCarrier.size() < 10) {
                        landingQueue.pop();
                        aircraftCarrier.push_back(plane);
                        cout << "Plane " << plane << " is landing..." << endl;
                        cout << "\nCurrent planes on carrier: " << aircraftCarrier.size() << " [ ";
                        for (size_t i = 0; i < aircraftCarrier.size(); ++i) {
                            cout << aircraftCarrier[i] << " ";
                        }
                        cout << "]\n" << endl;
                        actionCount++;
                    }
                }
            }
        }
        this_thread::sleep_for(chrono::milliseconds(500 + rand() % actionTime));
    }
}
```

Funkcja odpowiedzialna za synchronizację procesów lądowania i startowania w zależności od aktualnej liczby samolotów na lotniskowcu.

Działanie funkcji:

1. **Inicjalizacja generatora liczb pseudolosowych:**
   Funkcja rozpoczyna od inicjalizacji generatora liczb pseudolosowych za pomocą `srand`.

2. **Wejście w pętlę:**
   Funkcja wchodzi w pętlę, która kontynuuje działanie, dopóki funkcja `shouldExit()` nie zwróci `true`.

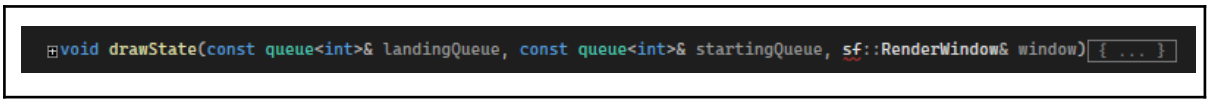
3. **Sprawdzanie, czy liczba samolotów na lotniskowcu jest mniejsza niż wartość zmiennej** `borderlineNumber`**:**
   Jeśli liczba samolotów na lotniskowcu jest mniejsza niż wartość zmiennej `borderlineNumber`, funkcja sprawdza, czy kolejka do lądowania nie jest pusta. Jeśli tak, to wybiera samolot z kolejki do lądowania i dodaje go do lotniskowca, jeśli na lotniskowcu jest mniej niż 10 samolotów. Jeśli kolejka do lądowania jest pusta, funkcja sprawdza, czy kolejka do startu nie jest pusta. Jeśli tak, to wybiera samolot z kolejki do startu i usuwa go z lotniskowca.

4. **Sprawdzanie, czy liczba samolotów na lotniskowcu jest większa lub równa wartości zmiennej** `borderlineNumber`**:**
   Jeśli liczba samolotów na lotniskowcu jest większa lub równa wartości zmiennej `borderlineNumber`, funkcja najpierw sprawdza, czy kolejka do startu nie jest pusta. Jeśli tak, to wybiera samolot z kolejki do startu i usuwa go z lotniskowca. Jeśli kolejka do startu jest pusta, funkcja sprawdza, czy kolejka do lądowania nie jest pusta. Jeśli tak, to wybiera samolot z kolejki do lądowania i dodaje go do lotniskowca, jeśli na lotniskowcu jest mniej niż 10 samolotów.

5. **Wyświetlanie informacji o stanie lotniskowca:**
   Po każdym lądowaniu lub starcie samolotu, funkcja wyświetla informacje o aktualnym stanie lotniskowca.

6. **Opóźnienie:**
   Na koniec każdej iteracji pętli funkcja wprowadza losowe opóźnienie, które jest zależne od zmiennej `actionTime`, ale nie krótsze niż 500 milisekund.

Funkcja drawState



Funkcja odpowiedzialna za wizualizację graficzną oraz kontrolowanie działania programu symulacyjnego przez użytkownika.

Działanie funkcji:

- **Wizualizacja aktualnego stanu lotniskowca:**
  Rysowanie aktualnego stanu symulacji na ekranie, co pozwala użytkownikowi monitorować i zrozumieć, co dzieje się na lotniskowcu.

- **Wizualizacja kolejek:**
  Kolejki do startu i lądowania są wizualnie przedstawione, co pozwala zobaczyć, które samoloty czekają na swoją kolej.

- **Rysowanie elementów interfejsu:**
  Dodatkowe elementy, takie jak przyciski, strzałki i linie, pomagają użytkownikowi w interakcji z symulacją.

- **Kontrola parametrów czasowych:**
  Umożliwia kontrolowanie czasu losowania dla kolejek do lądowania, startu i akcji. Użytkownik może dostosować te parametry w trakcie działania symulacji.

- **Kontrola ilości akcji programu:**
  Umożliwia kontrolowanie ilości akcji wykonanych przez program.

- **Kontrola progu priorytetowania**
  Umożliwia dynamiczne kontrolowanie liczby samolotów poniżej której priorytet mają samoloty lądujące.

Funkcja main

```cpp
int main() {
    srand(static_cast<unsigned>(time(nullptr)));

    sf::RenderWindow window(sf::VideoMode(626, 900), "Symulacja Lotniskowca", sf::Style::Titlebar | sf::Style::Close);

    sf::Image icon;
    icon.loadFromFile("images/plane.png");
    window.setIcon(icon.getSize().x, icon.getSize().y, icon.getPixelsPtr());

    texture.loadFromFile("images/background_water.png");
    sprite.setTexture(texture);

    texture2.loadFromFile("images/bottom.png");
    sprite2.setTexture(texture2);
    sprite2.setPosition(0, 485);

    thread landingThread(landing);
    thread startingThread(starting);
    thread actionThread(action);

    while (window.isOpen()) {
        sf::Event event;
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed) {
                window.close();
            }
            else if (event.type == sf::Event::MouseButtonPressed && event.mouseButton.button == sf::Mouse::Left) {
                sf::Vector2i mousePosition = sf::Mouse::getPosition(window);

                if (landUP.getGlobalBounds().contains(static_cast<float>(mousePosition.x), static_cast<float>(mousePosition.y))) {
                    if (landingTime < 3500) {
                        landingTime += 500;
                    }
                }
                else if (landDOWN.getGlobalBounds().contains(static_cast<float>(mousePosition.x), static_cast<float>(mousePosition.y))) {
                    if (landingTime > 500) {
                        landingTime -= 500;
                    }
                }
                else if (startUP.getGlobalBounds().contains(static_cast<float>(mousePosition.x), static_cast<float>(mousePosition.y))) {
                    if (startingTime < 3500) {
                        startingTime += 500;
                    }
                }
                else if (startDOWN.getGlobalBounds().contains(static_cast<float>(mousePosition.x), static_cast<float>(mousePosition.y))) {
                    if (startingTime > 500) {
                        startingTime -= 500;
                    }
                }
                else if (actionUP.getGlobalBounds().contains(static_cast<float>(mousePosition.x), static_cast<float>(mousePosition.y))) {
                    if (actionTime < 2500) {
                        actionTime += 500;
                    }
                }
                else if (actionDOWN.getGlobalBounds().contains(static_cast<float>(mousePosition.x), static_cast<float>(mousePosition.y))) {
                    if (actionTime > 500) {
                        actionTime -= 500;
                    }
                }
                else if (arrowUp.getGlobalBounds().contains(static_cast<float>(mousePosition.x), static_cast<float>(mousePosition.y))) {
                    if (action_to_stop < 150 && !shouldExit()) {
                        action_to_stop += 5;
                    }
                }
                else if (arrowDown.getGlobalBounds().contains(static_cast<float>(mousePosition.x), static_cast<float>(mousePosition.y))) {
                    if (action_to_stop - 5 >= actionCount) {
                        action_to_stop -= 5;
                    }
                }
                else if (arrowUp2.getGlobalBounds().contains(static_cast<float>(mousePosition.x), static_cast<float>(mousePosition.y))) {
                    if (borderlineNumber < 9 && !shouldExit()) {
                        borderlineNumber++;
                    }
                }
                else if (arrowDown2.getGlobalBounds().contains(static_cast<float>(mousePosition.x), static_cast<float>(mousePosition.y))) {
                    if (borderlineNumber > 1 && !shouldExit()) {
                        borderlineNumber--;
                    }
                }
            }
        }

        {
            lock_guard<mutex> lock(aircraftMutex);
            drawState(landingQueue, startingQueue, window);
        }

        this_thread::sleep_for(chrono::milliseconds(50));
    }

    landingThread.join();
    startingThread.join();
    actionThread.join();

    return 0;
}
```

Funkcja główna odpowiedzialna za obsługę pozostałych funkcji programu.

Działanie funkcji:

1. **Inicjalizacja generatora liczb pseudolosowych:**
   Funkcja rozpoczyna od inicjalizacji generatora liczb pseudolosowych za pomocą `srand`.

2. **Tworzenie okna SFML:**
   Tworzenie okna SFML o określonych parametrach oraz ładuje wykorzystane ikony i tekstury.

3. **Tworzenie wątków:**
   Tworzenie trzech wątków, które wykonują funkcje `landing`, `starting` i `action`.

4. **Główna pętla programu:**
   Funkcja wchodzi w główną pętlę programu, która działa, dopóki okno SFML jest otwarte. W pętli ta funkcja obsługuje zdarzenia SFML, aktualizuje i rysuje stan symulacji.

5. **Obsługa zdarzeń SFML:**
   W pętli zdarzeń funkcja obsługuje różne zdarzenia, takie jak zamknięcie okna, naciśnięcie przycisku myszy itp. W zależności od zdarzenia, funkcja może modyfikować różne zmienne, takie jak `landingTime`, `startingTime`, `actionTime`, `action_to_stop` i `borderlineNumber`.

6. **Aktualizacja i rysowanie stanu:**
   Funkcja aktualizuje i rysuje stan symulacji za pomocą funkcji `drawState`.

7. **Dołączanie wątków:**
   Po zamknięciu okna, funkcja dołącza wątki z powrotem do głównego wątku.

8. **Zwracanie 0:**
   Na koniec funkcja zwraca 0, co oznacza, że program zakończył działanie poprawnie.

7. Interfejs programu: