МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по практической работе №3
по дисциплине «Интеллектуальные системы»
Тема: «Управление несколькими игроками»

Студент гр. 8382 _____ Мирончик П.Д.

Преподаватель _____ Беляев С.А.

Санкт-Петербург
2022

ЦЕЛЬ РАБОТЫ

Целью работы является решение задачи координации действий автономных агентов с использованием деревьев решений.

Для достижения поставленной цели необходимо решить следующие задачи:

- разработать и подключить механизм обработки деревьев решений;
- разработать деревья для управления агентами.

ЗАДАНИЕ

Необходимо разработать с использованием деревьев решений две программы, имитирующие игрока и вратаря виртуального футбола.

Программа игрока должна решать следующие задачи:

1. Формирование «звена» из двух (трех) игроков. Для звена из двух игроков один впереди, второй чуть сзади слева, для трех игроков: один впереди, двое сзади (справа и слева). Решение о месте игрока в звене должно выбираться автономно каждым игроком.

2. Движение игроков в составе звена по заданному маршруту (последним пунктом маршрута – забивание гола в ворота справа).

Программа вратаря должна решать следующие задачи:

1. Защита ворот на правой половине поля.

2. Отбивание kick и поимка мяча catch.

ХОД РАБОТЫ

Для выполнения задания была реализована следующая машина состояний:

```
class StateTree {
```

```
    constructor(manager) {
        this.manager = manager
        this.mem = new Memory()
        this.states = {
            "init": (t) => CommonStates.stateInit(t),
            "find_ball": (t) => CommonStates.stateFindBall(t),
            "dribble": (t) => CommonStates.stateDribble(t),
            "move": (t) => CommonStates.stateMove(t),

            "team_attack": (t) =>
AttackerTeamStates.stateTeamAttack(t),
            "team_dribble": (t) =>
AttackerTeamStates.stateTeamDribble(t),
            "team_follow": (t) =>
AttackerTeamStates.stateTeamFollow(t),

            "select_goal": (t) => GoalStates.stateSelectGoal(t),
            "process_goal": (t) =>
GoalStates.stateProcessGoal(t),

            "goalie": (t) => GoalieStates.stateGoalie(t),
            "goalie_intercept": (t) =>
GoalieStates.stateGoalieIntercept(t),
            "goalie_kick_out": (t) =>
GoalieStates.stateGoalieKickOut(t),
            "goalie_catch": (t) =>
GoalieStates.stateGoalieCatch(t),
        }
        this.data = {}
    }

    makeCmd() {
        this._cmd = null
        this.callState("init")
        return this._cmd
    }

    callState(state) {
        this.states[state](this)
    }

    finish(cmd) {
        this._cmd = cmd
    }
}
```

Здесь *makeCmd* – метод, который запускает дерево состояний и формирует команду для отправки на сервер, *callState* – метод, запускающий очередное состояние по его названию, *finish* – метод, устанавливающий команду для отправки.

Переключение между состояниями осуществляется путем вызова следующего состояния из предыдущего, например:

```
    stateInit(tree) {
        let manager = tree.manager
        let data = tree.data
        let ball = manager.ball
        let firstVisible = -1;
        for (let i = 0; ;) {
            if (!ball) break;
            if (ball.visible) {
                firstVisible = i;
                break;
            }
            i++
            ball = ball.old
        }

        data.params = manager.agent.params
        data.agent = manager.agent

        if (firstVisible == -1 || firstVisible > 10) {
            return tree.callState("find_ball")
        }
        data.ball = manager.ball
        data.intercepts = manager.analyzeBallIntercepts()
        data.estimatedBallCoords = ball?.coords
        return tree.callState("select_goal")
    }
```

Здесь из состояния *init* реализованы два перехода, в состояние *find_ball* и в состояние *select_goal*. Из минусов подобной реализации можно отметить вероятность ошибки, когда после вызова *callState* не выполняется завершение состояния, однако такой подход позволяет реализовать максимально гибкую структуру, где возможны сложные условия перехода по состояниям, основанные на, например, анализе данных прошлых тактов.

Создан также класс, позволяющих хранить «устаревающие» данные, возраст которых увеличивается на каждом такте:

```
class Memory {
    constructor() {
        this._data = new Map()
    }

    increaseAge() {
        for (let key of this._data.keys()) {
            this._data.get(key).age++
```

```
        }
    }

    updateValue(key, value) {
        this._data.set(key, {
            value: value,
            age: 0
        })
    }

    getValue(key) {
        return this._data.get(key)?.value
    }

    getAge(key) {
        return this._data.get(key)?.age ?? 10000
    }

    has(key) {
        return this._data.has(key)
    }

    delete(key) {
        this._data.delete(key)
    }
}
```

Для более точного контроля мяча при возможности рассчитывается его скорость, исходя из получаемых *distChange* и *dirChange* параметров. В дальнейшем каждый игрок вычисляет положение мяча в следующих тактах и пытается выбрать оптимальный путь перехвата мяча с учетом расчетной скорости перемещения 0.3 от максимальной. Такая маленькая скорость выбрана для того, чтобы скорректировать потери скорости от возможных поворотов игрока. При этом также учитывается возможность перехвата мяча другими союзными игроками: если какой-то игрок из своей команды может эффективно перехватить раньше, чем сам агент, то перехват мяча агентом не выполняется.

С использованием вышеописанных средств реализовано дерево решений для игроков нападения и вратаря.

Игроки нападения действуют, выполняя определенные цели. Ими может быть либо ведение мяча в составе группы до определенной точки, либо нападение на ворота соперника. Нападение на ворота представляет

собой то же ведение мяча до ворот, но, когда расстояние до ворот становится менее чем 25, выполняется удар по воротам с максимальной силой.

В задачу вратаря входит отслеживание позиции мяча. Если мяч находится далее, чем в 10 единицах от вратаря, то задачей вратаря становится перемещение на позицию таким образом, чтобы вратарь оказался между центром ворот и мячом. Если расстояние менее 10 или вратарь может перехватить мяч раньше, чем противник, то выполняется перехват мяча. Если вратарь может пнуть мяч, то выполняется удар в сторону центра поля. Если вратарь может поймать мяч, то выполняется команда catch. При конфликте между поимкой мяча и ударом приоритет отдается удару по мячу.

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы была реализована мультиагентная система, состоящая из трех игроков нападения и одного вратаря в команде соперника. Игроки нападения выполняют перемещение в составе группы по заранее заданным точкам, выполняя при этом ведение мяча. После перемещения по маршруту выполняется нападение на ворота соперника. Вратарь при этом осуществляет защиту ворот, пытаясь по возможности поймать или ударить по мячу, либо подойти к мячу на расстояние удара, если тот находится недостаточно близко.