ДОДАТОК А
СЛАЙДИ ПРЕЗЕНТАЦІЇ

ХНУРЕ
Кафедра ПІ
Атестаційна робота магістра

Дослідження методів формалізації для ігрових ситуацій

Виконав ст. гр. ПЗСм-18-1 Шутєєв І.В.
Керівник к.т.н., доц. Каук І.В.

Об'єкт дослідження – ігровий процес.

Предмет дослідження – моделювання ігрових ситуацій.

Задачі дослідження:

- порівняння моделей формалізації ігрових ситуацій;
- обрання найбільш відповідних моделей;
- отримання необхідних вхідних даних для генерування моделей;
- вибір метрик для порівняння моделей;
- програмна реалізація обраних моделей;
- порівняння отриманих результатів.

2

АКТУАЛЬНІСТЬ РОБОТИ

- залежність у командних ігрових ситуаціях результатів від багатьох параметрів (у тому числі від особливих параметрів гравців);
- відсутність загально затверджених моделей, що однозначно дозволяють передбачити вірогідність результату гри;
- складність в отримані реальних даних для моделей, які б дозволили тренеру визначати траєкторію розвитку кожного з гравців.

3

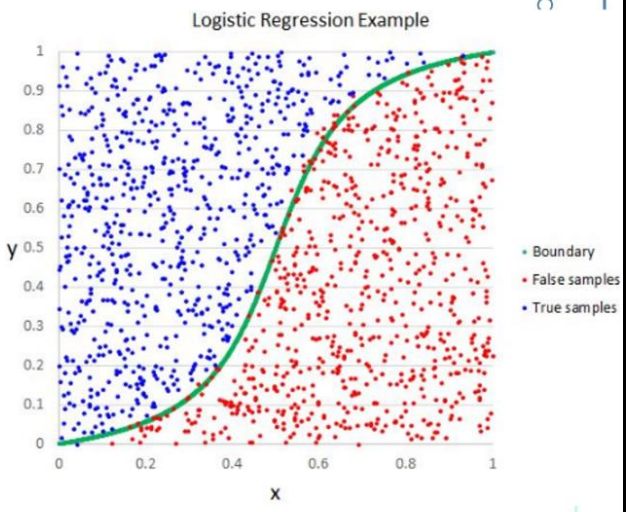
ОСНОВНІ ВИДИ КОМП'ЮТЕРНОГО МОДЕЛЮВАННЯ:

- фізичне моделювання; ✗
- динамічне або чисельне моделювання; ✗
- імітаційне моделювання; ✓
- статистичне моделювання; ✓
- інформаційне моделювання. ✗

4

СТАТИСТИЧНА МОДЕЛЬ

$$F(X) = \frac{1}{1 + e^{-\theta X}}$$



де $\theta = (\theta_0, \theta_1, \dots, \theta_n)$ – коефіцієнти моделі, $\theta X = \theta_0 + \theta_1 \cdot x_1 + \cdots + \theta_n \cdot x_n$.

5

МІНІМІЗАЦІЯ ФУНКЦІЇ ЛОГІСТИЧНОЇ РЕГРЕСІЇ

$$J(\theta) = \sum_{i=1}^{m} (Y_i \cdot \log F(X_i) + (1 - Y_i) \cdot \log(1 - F(X_i)))$$

де $F(X)$ – значення логістичної регресії.

$$\theta_j = \theta_j - \alpha \sum_{i=1}^{m} (F(X_i) - Y_i) \cdot x_j^i,$$

де x_j^i – параметр під номером j з набору X_i.

6

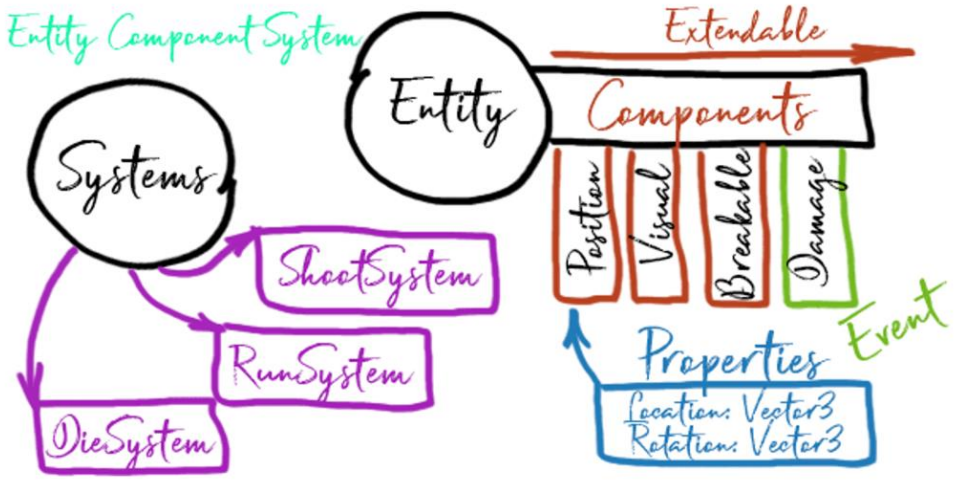
ІМІТАЦІЙНА МОДЕЛЬ

Імітаційне моделювання полягає в описі правил та логіки дій гравців під час ігрового процесу. Приклади правил:

- ЯКЩО відсоток влучання високий ТА немає захисника, ТОДІ кидок у кошик;
- ЯКЩО гравець без м'яча ТА з ним немає захисника, ТОДІ пас гравцю;
- ЯКЩО захисник далеко від свого гравця, ТОДІ рух до гравця;
- ЯКЩО кидок завершився, ТОДІ кінець гри.

7

ECS



8

ТЕХНОЛОГІЇ

Microsoft .NET

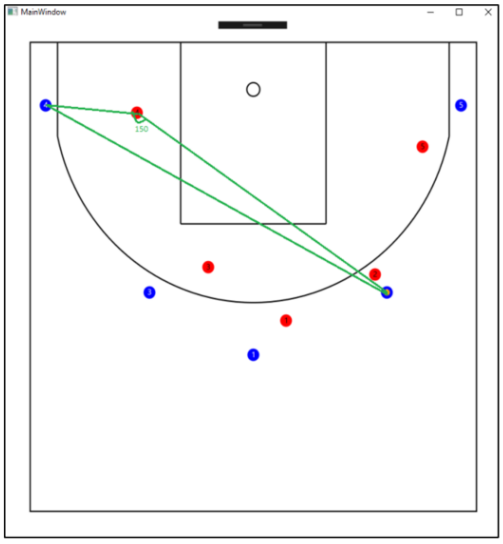
Accord Framework .NET

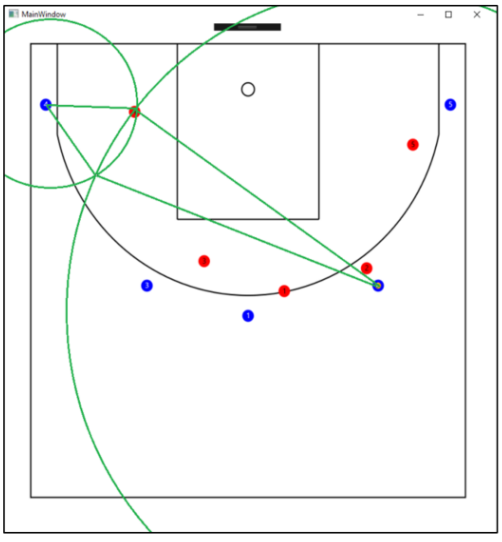
WPF

9

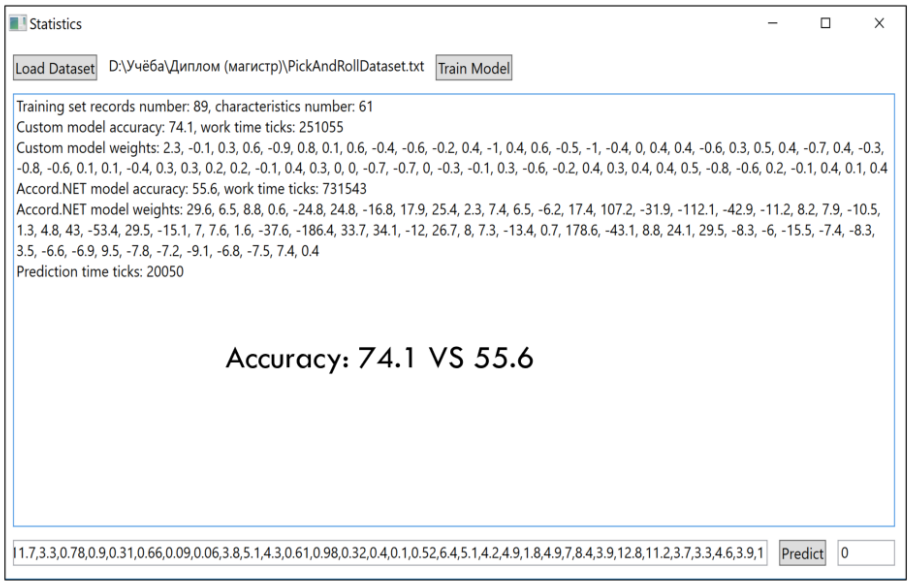
Логіка розташування захисників

10



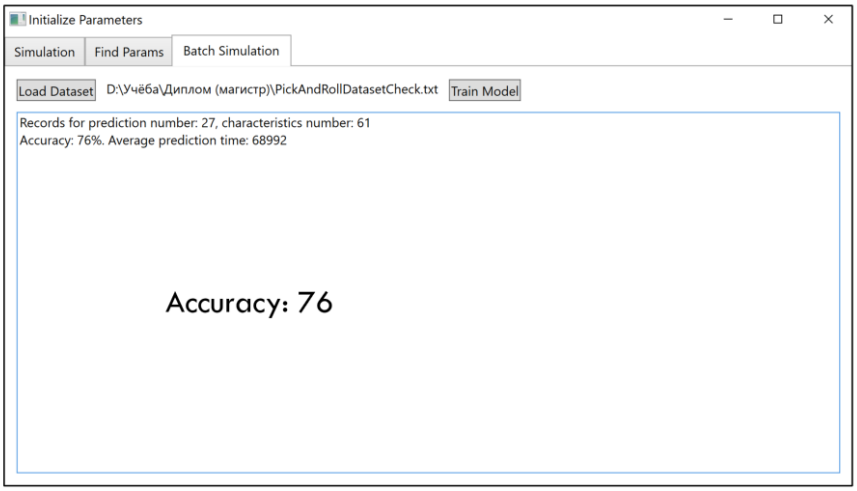


Результат роботи статистичної моделі



11

Результат роботи імітаційної моделі



12

Порівняння моделей

	Імітаційний метод	Статистичний метод
Необхідність навчання	ні	так
Необхідність датасету	ні	так
Необхідність ретельного підбору характеристик моделі	ні	так
Необхідність завдання правил поведінки об'єктів	так	ні
Швидкість передбачення	$O(R)$	$O(n)$
Залежність алгоритму від предметної області	так	ні
Точність (для обраної предметної області)	76%	74%

13

ВИСНОВКИ

14

В результаті роботи були порівняні моделі формалізації ігрових ситуацій. Найбільш відповідними були обрані імітаційна та статистична модель. Були сформовані датасети для навчання статистичної моделі та обрані наступні метрики для порівняння моделей:

- швидкість навчання;
- швидкість передбачення;
- точність.

Програмна реалізація обох моделей показала, що імітаційна модель не потребує навчання, статистична модель дає точний результат лише при наявності гарного датасету із точно підібраними характеристиками. Реалізація алгоритму статистичної моделі не залежить від предметної області. На одному й тому ж датасеті імітаційна модель показала точність 76%, а статистична модель – 74%.

Результати дослідження опубліковані на 15-ій міжнародній науковій конференції «Science And Society».

ДОДАТОК Б

ПРИКЛАДИ ПРОГРАМНОГО КОДУ

```
using System;
using System.Linq;
using GameSituationsStatistics.Models;

namespace GameSituationsStatistics.Prediction.LogisticRegression
{
    public class LogisticRegressionService : IPredictionService
    {
        private const double tolerance = 1e-4;
        private const int testSetPercentage = 70;

        public TrainedModel TrainModel(TrainingSet set)
        {
            if (set.X.Length != set.Y.Length)
            {
                throw new ArgumentException("Training set: input matrix lenght not equal to
output lenght.");
            }

            var m = set.X.Length;
            var n = set.X[0].Length;

            var averages = new double[n];
            var ranges = new double[n];

            var x = new double[m][];

            for (var j = 0; j < n; j++)
            {
                var max = set.X[0][j];
                var min = set.X[0][j];

                for (var i = 0; i < m; i++)
                {
                    averages[j] += set.X[i][j];

                    if (set.X[i][j] > max)
                    {
                        max = set.X[i][j];
                    }

                    if (set.X[i][j] < min)
                    {
                        min = set.X[i][j];
                    }
                }

                averages[j] /= m;
                ranges[j] = max - min;
            }

            for (var i = 0; i < m; i++)
            {
                x[i] = new double[n];

                for (var j = 0; j < n; j++)
                {
                    x[i][j] = (set.X[i][j] - averages[j]) / ranges[j];
                }
```

```
            }

            var trainSet = x.Take(x.Length * testSetPercentage / 100).ToArray();
            var testSet = x.Skip(trainSet.Length).ToArray();

            var trainSetY = set.Y.Take(trainSet.Length).ToArray();
            var testSetY = set.Y.Skip(trainSet.Length).ToArray();

            m = trainSet.Length;

            var thetas = new double[n + 1];
            var nextThetas = new double[n + 1];
            var alpha = 0.001;

            double cost = 0;
            double newCost = GetCostFunctionValue(nextThetas, trainSet, trainSetY);

            do
            {
                cost = newCost;

                for (var i = 0; i < nextThetas.Length; i++)
                {
                    thetas[i] = nextThetas[i];
                }

                var functionDifferences = GetNewFunctionDifferences(thetas, trainSet,
trainSetY);

                double sumTheta0 = 0;

                for (var i = 0; i < m; i++)
                {
                    sumTheta0 += functionDifferences[i];
                }

                nextThetas[0] = thetas[0] - alpha * sumTheta0;

                for (int j = 0; j < n; j++)
                {
                    double sum = 0;

                    for (var i = 0; i < m; i++)
                    {
                        sum += functionDifferences[i] * trainSet[i][j];
                    }

                    nextThetas[j + 1] = thetas[j + 1] - alpha * sum;
                }

                newCost = GetCostFunctionValue(nextThetas, trainSet, trainSetY);

            } while (Math.Abs(cost - newCost) > tolerance);

            var testSetAccuracy = testSet
                .Select(s => GetFunctionResult(nextThetas, s) < 0.5 ? 0 : 1)
                .Zip(testSetY, (trainedResult, actualResult) => trainedResult == actualResult)
                .Where(isEqual => isEqual)
                .Count() / (double)testSetY.Length * 100;

            return new LogisticRegressionTrainedModel(nextThetas, averages, ranges,
testSetAccuracy);
        }
```

```
        private double GetFunctionResult(double[] thetas, double[] x)
        {
            if (thetas.Length != x.Length + 1)
            {
                throw new ArgumentException("Training set: thetas lenght does not correspond
to parameters length.");
            }

            double sum = thetas[0];

            for (int i = 0; i < x.Length; i++)
            {
                sum += x[i] * thetas[i + 1];
            }

            return 1 / (1 + Math.Pow(Math.E, -sum));
        }

        private double[] GetNewFunctionDifferences(double[] thetas, double[][] x, double[] y)
        {
            var functionDifferences = new double[y.Length];

            for (var i = 0; i < y.Length; i++)
            {
                functionDifferences[i] = GetFunctionResult(thetas, x[i]) - y[i];
            }

            return functionDifferences;
        }

        private double GetCostFunctionValue(double[] thetas, double[][] x, double[] y)
        {
            var m = y.Length;
            double sum = 0;

            for (int i = 0; i < m; i++)
            {
                var functionResult = GetFunctionResult(thetas, x[i]);
                sum += y[i] * Math.Log(functionResult) + (1 - y[i]) * Math.Log(1 -
functionResult);
            }

            return -sum / m;
        }
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using Autofac;
using GameSituationsStatistics.Core;
using GameSituationsStatistics.IoC;
```

```
using GameSituationsStatistics.Models;
using Microsoft.Win32;

namespace GameSituationsStatistics.UI
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        private IStatisticsService service;
        private TrainedModel trainedModel;
        private TrainingSet trainingSet;
        private TrainedModel TrainedModel
        {
            get { return trainedModel; }
            set
            {
                trainedModel = value;

                if (trainedModel == null)
                {
                    predictionPanel.Visibility = Visibility.Collapsed;
                }
                else
                {
                    predictionPanel.Visibility = Visibility.Visible;
                }
            }
        }
        private TrainingSet TrainingSet
        {
            get { return trainingSet; }
            set
            {
                trainingSet = value;

                if (trainingSet != null)
                {
                    fileNameTextBlock.Visibility = Visibility.Visible;
                    trainModelButton.Visibility = Visibility.Visible;

                }
                else
                {
                    fileNameTextBlock.Visibility = Visibility.Collapsed;
                    trainModelButton.Visibility = Visibility.Collapsed;
                }
            }
        }

        public MainWindow()
        {
            InitializeComponent();

            var builder = DependenciesRegistrator.Configure();
            service = builder.Resolve<IStatisticsService>();
        }

        private void LoadDataSetButton_Click(object sender, RoutedEventArgs e)
        {
            OpenFileDialog openFileDialog = new OpenFileDialog();

            if (openFileDialog.ShowDialog() == true)
```

```
            {
                LoadDataSet(openFileDialog.FileName);
            }
        }

        private void LoadDataSet(string pathToDataset)
        {
            TrainingSet = service.LoadTrainingSet(pathToDataset);

            outputTextBox.Text += $"Training set records number: {trainingSet.Y.Length},
characteristics number: {trainingSet.X[0].Length}\r\n";

            fileNameTextBlock.Text = pathToDataset;
        }

        private void TrainModel()
        {
            var date1Ticks = DateTime.Now.Ticks;
            var customtrainedModel = service.TrainCustomModel(TrainingSet);
            var date2Ticks = DateTime.Now.Ticks;
            TrainedModel = service.TrainAccordModel(TrainingSet);
            var date3Ticks = DateTime.Now.Ticks;

            // 1 ms = 10000 ticks
            outputTextBox.Text += $"Custom model accuracy: {customtrainedModel.Accuracy}, work
time ticks: {date2Ticks - date1Ticks}\r\n";
            outputTextBox.Text += $"Custom model weights: {string.Join(", ",
customtrainedModel.Coefficients.Select(x => Math.Round(x, 1)))}\r\n";
            outputTextBox.Text += $"Accord.NET model accuracy: {TrainedModel.Accuracy}, work
time ticks: {date3Ticks - date2Ticks}\r\n";
            outputTextBox.Text += $"Accord.NET model weights: {string.Join(", ",
TrainedModel.Coefficients.Select(x => Math.Round(x, 1)))}\r\n";
        }

        private void PredictNewValuesButtonClick(object sender, RoutedEventArgs e)
        {
            var input = newValuesInputTextBox.Text.Split(',', ';').Select(x =>
Convert.ToDouble(x)).ToArray();

            var date1Ticks = DateTime.Now.Ticks;
            var output = Math.Round(TrainedModel.PredictValue(input), 2);
            var date2Ticks = DateTime.Now.Ticks;

            newValuesOutputTextBox.Text = output.ToString();

            outputTextBox.Text += $"Prediction time ticks: {date2Ticks - date1Ticks}\r\n";
        }

        private void TrainModelButton_Click(object sender, RoutedEventArgs e)
        {
            TrainModel();
        }
    }
}

using BasketballGameSimulator.BaseEcsLogic;
using BasketballGameSimulator.Core.Components;
using BasketballGameSimulator.Core.Entities;
using BasketballGameSimulator.Core.Entities.Enums;
using BasketballGameSimulator.Core.Utils;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using System.Threading.Tasks;

namespace BasketballGameSimulator.Core.Systems
{
    public class OffenseSystem : GameSystem
    {
        private Ball ball;
        private List<OffensivePlayer> offensivePlayers;
        private List<DefensivePlayer> defensivePlayers;

        private Dictionary<CombinationType, Action> combinations = new
Dictionary<CombinationType, Action>();
        private CombinationType combination;
        private int combinationStep;

        public OffenseSystem(Ball ball, List<OffensivePlayer> offensivePlayers,
List<DefensivePlayer> defensivePlayers, CombinationType combination)
        {
            this.ball = ball;
            this.offensivePlayers = offensivePlayers;
            this.defensivePlayers = defensivePlayers;

            this.combination = combination;
            combinationStep = 1;

            combinations.Add(CombinationType.PickAndRoll, PickAndRoll);
            combinations.Add(CombinationType.KhnureNew, KhnureNewCombination);
        }

        public override void Process()
        {
            var playerWithBall = GetPlayerWithBall();

            // 1. If there is no plater with the ball, don't change anything.
            if (playerWithBall == null)
            {
                return;
            }

            var playerWithBallDistanceToRim = GetDistanceToRim(playerWithBall);
            var playerWithBallDistanceToDefender =
GetDistanceToClosestDefender(playerWithBall);

            // 2. If player with ball is under the rim and is uncontestant then shot.
            if (playerWithBallDistanceToRim < GameConsts.MIDRANGE_SHOT_RANGE
                && playerWithBallDistanceToDefender > GameConsts.CONTESTANT_SHOT_RANGE
                && playerWithBall.ShotComponent.LayupUncontestantShotPercentage >
GameConsts.LAYUP_PERCENTAGE_TO_SHOT)
            {
                Shoot(playerWithBall);
                return;
            }

            var playersWithoutBall = offensivePlayers.Where(x => x != playerWithBall);
            var playersWithoutBallDistancesToRim = playersWithoutBall.ToDictionary(x => x, x
=> GetDistanceToRim(x));
            var playersWithoutBallDistancesToDefender = playersWithoutBall.ToDictionary(x =>
x, x => GetDistanceToClosestDefender(x));
            var playersWithoutBallDistancesToPlayerWithBall =
playersWithoutBall.ToDictionary(x => x, x => GetDistanceBetweenEntities(x.MovementComponent,
playerWithBall.MovementComponent));

            // 3. If there is a free player without ball under the rim pass him the ball.
            foreach (var playerWithoutBall in playersWithoutBall)
```

```
            {
                if (playersWithoutBallDistancesToRim[playerWithoutBall] <
GameConsts.MIDRANGE_SHOT_RANGE
                && (playersWithoutBallDistancesToDefender[playerWithoutBall] -
GameConsts.CONTESTANT_SHOT_RANGE) /
GetPlayerDefender(playerWithoutBall).MovementComponent.Speed >
                    playersWithoutBallDistancesToPlayerWithBall[playerWithoutBall] /
GameConsts.PASS_SPEED)
                {
                    if ((playerWithBall.MovementComponent.X <= CourtSizes.RIM_CENTER_X &&
                        playerWithoutBall.MovementComponent.X <= CourtSizes.RIM_CENTER_X) ||
                        (playerWithBall.MovementComponent.X >= CourtSizes.RIM_CENTER_X &&
                        playerWithoutBall.MovementComponent.X >= CourtSizes.RIM_CENTER_X))
                    {
                        Pass(playerWithoutBall);
                        return;
                    }
                }
            }

            // 4. If player with ball is at the mid-range distance and he is uncontestant then
shot and his percantage is high then shoot.
            if (playerWithBallDistanceToRim < GameConsts.THREE_POINT_SHOT_RANGE
                && playerWithBallDistanceToDefender > GameConsts.CONTESTANT_SHOT_RANGE
                && playerWithBall.ShotComponent.MidRangeUncontestantShotPercentage >
GameConsts.MIDRANGE_PERCENTAGE_TO_SHOT)
            {
                Shoot(playerWithBall);
                return;
            }

            // 5. If there is a free player without ball at the mid-range distance and his
mid-range percentage is high pass him the ball.
            foreach (var playerWithoutBall in playersWithoutBall)
            {
                if (playersWithoutBallDistancesToRim[playerWithoutBall] <
GameConsts.THREE_POINT_SHOT_RANGE
                && (playersWithoutBallDistancesToDefender[playerWithoutBall] -
GameConsts.CONTESTANT_SHOT_RANGE) /
GetPlayerDefender(playerWithoutBall).MovementComponent.Speed >
                    playersWithoutBallDistancesToPlayerWithBall[playerWithoutBall] /
GameConsts.PASS_SPEED
                && playerWithoutBall.ShotComponent.MidRangeUncontestantShotPercentage >
GameConsts.MIDRANGE_PERCENTAGE_TO_SHOT)
                {
                    if ((playerWithBall.MovementComponent.X <= CourtSizes.RIM_CENTER_X &&
                        playerWithoutBall.MovementComponent.X <= CourtSizes.RIM_CENTER_X) ||
                        (playerWithBall.MovementComponent.X >= CourtSizes.RIM_CENTER_X &&
                        playerWithoutBall.MovementComponent.X >= CourtSizes.RIM_CENTER_X))
                    {
                        Pass(playerWithoutBall);
                        return;
                    }
                }
            }

            // 5. If player with ball is at the three point distance and he is uncontestant
then shot and his percantage is high then shoot.
            if (playerWithBallDistanceToRim >= GameConsts.THREE_POINT_SHOT_RANGE
                && playerWithBallDistanceToDefender > GameConsts.CONTESTANT_SHOT_RANGE
                && playerWithBall.ShotComponent.ThreePointUncontestantShotPercentage >
GameConsts.THREE_POINT_PERCENTAGE_TO_SHOT)
            {
                Shoot(playerWithBall);
```

```
                return;
            }

            // 6. If there is a free player without ball at the three point distance and his
three point percentage is high pass him the ball.
            foreach (var playerWithoutBall in playersWithoutBall)
            {
                if (playersWithoutBallDistancesToRim[playerWithoutBall] <
GameConsts.THREE_POINT_SHOT_RANGE
                && (playersWithoutBallDistancesToDefender[playerWithoutBall] -
GameConsts.CONTESTANT_SHOT_RANGE) /
GetPlayerDefender(playerWithoutBall).MovementComponent.Speed >
                    playersWithoutBallDistancesToPlayerWithBall[playerWithoutBall] /
GameConsts.PASS_SPEED
                && playerWithoutBall.ShotComponent.ThreePointUncontestantShotPercentage >
GameConsts.MIDRANGE_PERCENTAGE_TO_SHOT)
                {
                    if ((playerWithBall.MovementComponent.X <= CourtSizes.RIM_CENTER_X &&
                        playerWithoutBall.MovementComponent.X <= CourtSizes.RIM_CENTER_X) ||
                        (playerWithBall.MovementComponent.X >= CourtSizes.RIM_CENTER_X &&
                        playerWithoutBall.MovementComponent.X >= CourtSizes.RIM_CENTER_X))
                    {
                        Pass(playerWithoutBall);
                        return;
                    }
                }
            }

            combinations[combination]();
        }

        private void PickAndRoll()
        {
            var playerWithBall = GetPlayerWithBall();
            var playerWithBallDefender = GetPlayerDefender(playerWithBall);
            var pickPlayer = GetPlayerByNumber(5);

            if (pickPlayer == playerWithBall)
            {
                pickPlayer = GetPlayerByNumber(4);
            }

            if (combinationStep == 1)
            {
                pickPlayer.MovementComponent.ToX = playerWithBallDefender.MovementComponent.X;
                pickPlayer.MovementComponent.ToY =
                    playerWithBallDefender.MovementComponent.Y -
pickPlayer.MovementComponent.Radius - playerWithBall.MovementComponent.Radius;

                if (pickPlayer.MovementComponent.X == pickPlayer.MovementComponent.ToX &&
                    pickPlayer.MovementComponent.Y == pickPlayer.MovementComponent.ToY)
                {
                    combinationStep = 2;
                }
            }

            if (combinationStep == 2)
            {
                Dribble(playerWithBall, CourtSizes.RIM_CENTER_X, CourtSizes.RIM_CENTER_Y);

                if (playerWithBall.MovementComponent.Y <= pickPlayer.MovementComponent.Y)
                {
                    combinationStep = 3;
                }
```

```
    }

    if (combinationStep == 3)
    {
        pickPlayer.MovementComponent.ToX = CourtSizes.RIM_CENTER_X;
        pickPlayer.MovementComponent.ToY = CourtSizes.RIM_CENTER_Y;

        var playerWithBallDistanceToRim = GetDistanceToRim(playerWithBall);

        if (playerWithBallDistanceToRim < GameConsts.MIDRANGE_SHOT_RANGE)
        {
            Shoot(playerWithBall);
            combinationStep = 4;
        }
    }
}

private void KhnureNewCombination()
{
    var firstPlayer = GetPlayerByNumber(1);
    var secondPlayer = GetPlayerByNumber(2);
    var thirdPlayer = GetPlayerByNumber(3);
    var fourthPlayer = GetPlayerByNumber(4);
    var fifthPlayer = GetPlayerByNumber(5);

    var playerWithBall = GetPlayerWithBall();

    if (combinationStep == 1)
    {
        if (playerWithBall == null)
        {
            return;
        }
        else if (playerWithBall != firstPlayer)
        {
            Pass(firstPlayer);
        }
        else
        {
            Dribble(firstPlayer, 7.5, 9);

            secondPlayer.MovementComponent.ToX = 4;
            secondPlayer.MovementComponent.ToY = 6;

            thirdPlayer.MovementComponent.ToX = 5;
            thirdPlayer.MovementComponent.ToY = 2;

            fourthPlayer.MovementComponent.ToX = 4;
            fourthPlayer.MovementComponent.ToY = 7;

            fifthPlayer.MovementComponent.ToX = 10;
            fifthPlayer.MovementComponent.ToY = 3;

            if (AreAllPlayersStanding())
            {
                combinationStep = 2;
                return;
            }
        }
    }

    if (combinationStep == 2)
    {
        var secondPlayerDefender = GetPlayerDefender(secondPlayer);
```

```
                thirdPlayer.MovementComponent.ToX = secondPlayerDefender.MovementComponent.X;
                thirdPlayer.MovementComponent.ToY = secondPlayerDefender.MovementComponent.Y -
0.5;

                if (IsPlayerStanding(thirdPlayer))
                {
                    combinationStep = 3;
                    return;
                }
            }

            if (combinationStep == 3)
            {
                secondPlayer.MovementComponent.ToX = CourtSizes.RIM_CENTER_X;
                secondPlayer.MovementComponent.ToY = CourtSizes.RIM_CENTER_Y;

                thirdPlayer.MovementComponent.ToX = fourthPlayer.MovementComponent.X;
                thirdPlayer.MovementComponent.ToY = fourthPlayer.MovementComponent.Y + 0.5;

                if (IsPlayerStanding(thirdPlayer))
                {
                    Pass(thirdPlayer);
                    combinationStep = 4;

                    return;
                }
            }

            if (combinationStep == 4)
            {
                if (ball.BallState == BallState.Dribbled)
                {
                    Shoot(thirdPlayer);
                    combinationStep = 6;
                    return;
                }
                else
                {
                    secondPlayer.MovementComponent.ToX = fifthPlayer.MovementComponent.X + 1;
                    secondPlayer.MovementComponent.ToY = fifthPlayer.MovementComponent.Y;

                    thirdPlayer.MovementComponent.ToX = fourthPlayer.MovementComponent.X -
0.5;


                    if (IsPlayerStanding(thirdPlayer))
                    {
                        combinationStep = 5;
                        return;
                    }
                }
            }

            if (combinationStep == 5)
            {
                if (playerWithBall == thirdPlayer)
                {
                    Shoot(thirdPlayer);
                    combinationStep = 6;
                    return;
                }
            }
        }
```

```
        private void Pass(OffensivePlayer player)
        {
            ball.MovementComponent.ToX = player.MovementComponent.X;
            ball.MovementComponent.ToY = player.MovementComponent.Y;
            ball.MovementComponent.Speed = GameConsts.PASS_SPEED;
            ball.SetPassedBallState(player);
        }

        private void Shoot(OffensivePlayer playerWithBall)
        {
            var distanceToDefender = GetDistanceToClosestDefender(playerWithBall);

            var randomNumber = RandomGenerator.Generate();

            var distanceToRim = GetDistanceToRim(playerWithBall);

            if (distanceToRim < GameConsts.MIDRANGE_SHOT_RANGE)
            {
                if (distanceToDefender < GameConsts.CONTESTANT_SHOT_RANGE)
                {
                    ball.BucketComponent.IsShotSuccessful = randomNumber <
playerWithBall.ShotComponent.LayupContestantShotPercentage;
                }
                else
                {
                    ball.BucketComponent.IsShotSuccessful = randomNumber <
playerWithBall.ShotComponent.LayupUncontestantShotPercentage;
                }
            }
            else if (distanceToRim < GameConsts.THREE_POINT_SHOT_RANGE)
            {
                if (distanceToDefender < GameConsts.CONTESTANT_SHOT_RANGE)
                {
                    ball.BucketComponent.IsShotSuccessful = randomNumber <
playerWithBall.ShotComponent.MidRangeContestantShotPercentage;
                }
                else
                {
                    ball.BucketComponent.IsShotSuccessful = randomNumber <
playerWithBall.ShotComponent.MidRangeUncontestantShotPercentage;
                }
            }
            else
            {
                if (distanceToDefender < GameConsts.CONTESTANT_SHOT_RANGE)
                {
                    ball.BucketComponent.IsShotSuccessful = randomNumber <
playerWithBall.ShotComponent.ThreePointContestantShotPercentage;
                }
                else
                {
                    ball.BucketComponent.IsShotSuccessful = randomNumber <
playerWithBall.ShotComponent.ThreePointUncontestantShotPercentage;
                }
            }

            ball.MovementComponent.ToX = CourtSizes.RIM_CENTER_X;
            ball.MovementComponent.ToY = CourtSizes.RIM_CENTER_Y;
            ball.MovementComponent.Speed = GameConsts.SHOT_SPEED;

            ball.SetShotBallState();
        }

        private void Dribble(OffensivePlayer playerWithBall, double toX, double toY)
```

```
        {
            playerWithBall.MovementComponent.ToX = toX;
            playerWithBall.MovementComponent.ToY = toY;
            ball.MovementComponent.X = playerWithBall.MovementComponent.X;
            ball.MovementComponent.Y = playerWithBall.MovementComponent.Y;
            ball.MovementComponent.ToX = toX;
            ball.MovementComponent.ToY = toY;
            ball.MovementComponent.Speed = playerWithBall.MovementComponent.Speed;

            ball.SetDribbledBallState(playerWithBall);
        }

        private OffensivePlayer GetPlayerWithBall()
        {
            if (ball.DribbledBy != null)
            {
                return ball.DribbledBy;
            }

            foreach (var offensivePlayer in offensivePlayers)
            {
                if (Math.Abs(offensivePlayer.MovementComponent.X - ball.MovementComponent.X) <
GameConsts.ERROR
                    && Math.Abs(offensivePlayer.MovementComponent.Y -
ball.MovementComponent.Y) < GameConsts.ERROR)
                {
                    ball.SetDribbledBallState(offensivePlayer);
                }
            }

            return null;
        }

        private double GetDistanceToRim(OffensivePlayer player)
        {
            return Math.Sqrt(
                        Math.Pow(player.MovementComponent.X - CourtSizes.RIM_CENTER_X, 2) +
                        Math.Pow(player.MovementComponent.Y - CourtSizes.RIM_CENTER_Y, 2));
        }

        private double GetDistanceToClosestDefender(OffensivePlayer player)
        {
            var minDistance = double.MaxValue;

            foreach (var defensivePlayer in defensivePlayers)
            {
                var distance = Math.Sqrt(
                        Math.Pow(player.MovementComponent.X -
defensivePlayer.MovementComponent.X, 2) +
                        Math.Pow(player.MovementComponent.Y -
defensivePlayer.MovementComponent.Y, 2));

                if (distance < minDistance)
                {
                    minDistance = distance;
                }
            }

            return minDistance;
        }

        private double GetDistanceBetweenEntities(MovementComponent movementComponent1,
MovementComponent movementComponent2)
        {
```

```
            return Math.Sqrt(
                    Math.Pow(movementComponent1.X - movementComponent2.X, 2) +
                    Math.Pow(movementComponent1.Y - movementComponent2.Y, 2));
        }

        private OffensivePlayer GetPlayerByNumber(int number)
        {
            return offensivePlayers.FirstOrDefault(x => x.PlayerComponent.Number == number);
        }

        private DefensivePlayer GetPlayerDefender(OffensivePlayer player)
        {
            return defensivePlayers.FirstOrDefault(x => x.DefenseComponent.GuardedPlayer ==
player);
        }

        private bool IsPlayerStanding(OffensivePlayer player)
        {
            return !player.MovementComponent.ToX.HasValue ||
                !player.MovementComponent.ToY.HasValue ||
                (Math.Abs(player.MovementComponent.ToX.Value - player.MovementComponent.X) <
GameConsts.ERROR &&
                        (Math.Abs(player.MovementComponent.ToY.Value -
player.MovementComponent.Y) < GameConsts.ERROR));
        }

        private bool AreAllPlayersStanding()
        {
            foreach(var player in offensivePlayers)
            {
                if (!IsPlayerStanding(player))
                {
                    return false;
                }
            }

            return true;
        }
    }
}
```

ДОДАТОК В

ТЕЗИ НАУКОВО-ТЕХНІЧНОЇ КОНФЕРЕНЦІЇ «SCIENCE AND SOCIETY»

ДОСЛІДЖЕННЯ МЕТОДІВ ФОРМАЛІЗАЦІЇ ДЛЯ ІГРОВИХ СИТУАЦІЙ

ШУТЄЄВ І.В.

студент групи ПЗСм-18-1,

кафедра Програмної Інженерії, факультет Комп'ютерних Наук,

Харківський національний університет радіоелектроніки

м. Харків, Україна

Гра – діяльність людини з моделювання іншого виду діяльності з розважальною чи навчальною метою. Гра проходить за певними правилами, які встановлюють обмеження на дії гравця [1].

У грі може брати участь один або кілька гравців. У грі, що розрахована на одну людину, гравець ставить перед собою складне завдання, яке намагається розв'язати, дотримуючись встановлених для себе правил. Ігри для багатьох гравців часто ставлять перед собою завдання виграти, здобути перемогу над іншими, тобто гра є змаганням. У іграх з елементами змагання гравці можуть утворювати команди – колективи, які намагаються досягти спільної мети. Відповідні ігри називаються командними [2].

Метою моделювання, у тому числі і ігор, є здобуття, обробка, представлення і використання інформації про об'єкти, які взаємодіють між собою і зовнішнім середовищем [3]. Завдяки моделюванню ігор та методів прийняття рішень в них можна зрозуміти яка тактика є вигршною або програшною.

Моделювання – це процес створення моделі для подальшого дослідження об’єкта (процесу, явища). Інформаційна модель – це опис важливих для певного дослідження властивостей об’єкта. Комп'ютерна модель – це інформаційна модель, реалізована за допомогою програмного середовища.

Об’єктом дослідження даної роботи є процес гри. Предметом дослідження є методи моделювання ігрових ситуацій.

Створення різних програмних моделей ігрових процесів допоможе наглядно оцінити та порівняти характеристики ігор серед яких і ймовірність перемоги або програшу від використовування певної стратегії. Така програма допоможе обирати користувачу необхідну тактику залежно від ігрової ситуації.

Задачею дослідження є порівняння моделей ігрових процесів з метою обрання кращої в залежності від контексту.

У загальному випадку поведінка досліджуваної системи S описується законом функціонування $Y(t) = F(X, V, H, t)$, де $X = (x_1, x_2, \dots, x_n)$, – вектор вхідних впливів (стимулів), $Y = (y_1, y_2, \dots, y_m)$ – вектор вихідних сигналів (відгуків, реакцій), $V = (v_1, v_2, \dots, v_k)$ – вектор впливів зовнішнього середовища, $H = (h_1, h_2, \dots, h_l)$ – вектор власних параметрів системи [4]. Закон функціонування може мати вигляд словесного правила, таблиці, алгоритму, функції, сукупності логічних умов і т.д. У разі, коли закон функціонування містить час, говорять про динамічні моделі та системи. Наприклад, розгін і гальмування асинхронного двигуна, перехідний процес в ланцюзі, що містить конденсатор, функціонування обчислювальної мережі, системи масового обслуговування. У всіх цих випадках стан системи, а значить і її моделі, змінюється з плином часу.

Якщо поведінка системи описується законом $Y = F(X, V, H)$, що не містить час t явно, то мова йде про статичні моделі та системи, вирішення стаціонарних задач і т.д. Наприклад: розрахунок нелінійного ланцюга постійного струму, знаходження стаціонарного розподілу температури в стержні при постійних температурах його кінців, форми пружної плівки, натягнутої на каркас, профілю швидкостей в сталому перебігу в'язкої рідини і т.д.

Функціонування системи можна розглядати як послідовну зміну станів $q_1(t), q_2(t), \dots, q_r(t)$, яким відповідають деякі точки в багатовимірному фазовому просторі. Безліч всіх точок $A = \{a_1, a_2, \dots, a_s\}$, що відповідають всіляким станам системи, називають простором станів об'єкта (або моделі). Кожній реалізації процесу відповідає одна фазова траєкторія, що проходить

через деякі точки з множини A. Якщо математична модель містить елемент випадковості, то виходить стохастична комп'ютерна модель. В окремому випадку, коли параметри системи і зовнішні впливи однозначно визначають вихідні сигнали, говорять про детерміновану модель.

Під комп'ютерним моделюванням в найширшому сенсі будемо розуміти процес створення і дослідження моделей за допомогою комп'ютера. Виділяють наступні види моделювання:

– фізичне моделювання [4]. Комп'ютер – частина експериментальної установки або тренажера, він сприймає зовнішні сигнали, здійснює відповідні розрахунки і видає сигнали, що керують різними маніпуляторами. Наприклад, навчальна модель літака, що представляє собою кабіну, встановлену на відповідних маніпуляторах, з'єднаних з комп'ютером, який реагує на дії пілота і змінює нахил кабіни, показання приладів, вид з ілюмінатора і т.д., імітуючи політ реального літака;

– динамічне або чисельне моделювання, що припускає чисельне рішення системи алгебраїчних і диференціальних рівнянь способами обчислювальної математики і проведення обчислювального експерименту при різних параметрах системи, початкових умовах і зовнішніх впливах [5]. Використовується для моделювання різних фізичних, біологічних, соціальних та інших явищ: коливання маятника, поширення хвилі, зміна чисельності населення, популяції цього виду тварин і т.д.;

– імітаційне моделювання полягає в створенні комп'ютерної програми (або пакета програм), що імітує поведінку складної технічної, економічної чи іншої системи на ЕОМ з необхідною точністю [6]. Імітаційне моделювання передбачає формальний опис логіки функціонування досліджуваної системи з плином часу, який враховує суттєві взаємодії її компонентів і забезпечує проведення статистичних експериментів. Об'єктно-орієнтовані комп'ютерні симуляції використовуються для дослідження поведінки економічних, біологічних, соціальних та інших систем, для створення комп'ютерних ігор, так

званого «віртуального світу», навчальних програм і анімацій. Наприклад, модель технологічного процесу, аеродрому, деякою галузі виробництва і т.д.;

– статистичне моделювання використовується для вивчення стохастичних систем і складається в багаторазовому проведенні випробувань з подальшою статистичною обробкою отриманих результатів [7]. Подібні моделі дозволяють вивчати поведінку всіляких систем масового обслуговування, багатопроцесорних систем, інформаційно-обчислювальних мереж, різних динамічних систем, на які впливають випадкові чинники. Статистичні моделі застосовуються при вирішенні імовірнісних задач, а також при обробці великих масивів даних (інтерполяція, екстраполяція, регресія, кореляція, розрахунок параметрів розподілу і т.д.). Вони відрізняються від детермінованих моделей, використання яких передбачає чисельне рішення систем алгебраїчних або диференціальних рівнянь, або заміну досліджуваного об'єкта детермінованим автоматом;

– інформаційне моделювання полягає в створенні інформаційної моделі, тобто сукупності спеціальним чином організованих даних (знаків, сигналів), що відображають найбільш суттєві сторони досліджуваного об’єкта. Розрізняють наочні, графічні, анімаційні, текстові, табличні інформаційні моделі. До них відносяться всілякі схеми, графи, графіки, таблиці, діаграми, малюнки, анімації, виконані на ЕОМ, в тому числі цифрова карта зоряного неба, комп’ютерна модель земної поверхні і т.д;

– моделювання знань передбачає побудову системи штучного інтелекту, в основі якої лежить база знань деякої предметної області (частини реального світу). Бази знань складаються з фактів (даних) і правил. Наприклад, комп’ютерна програма, яка вміє грати в шахи, повинна оперувати інформацією про «здібності» різних шахових фігур і «знати» правила гри. До даного виду моделей відносять семантичні мережі, логічних моделі знань, експертні системи, логічні ігри і т.д. Логічні моделі використовуються для представлення знань в експертних системах, для створення систем штучного інтелекту, здійснення логічного висновку, доведення теорем, математичних перетворень,

побудови роботів, використання природної мови для спілкування з ЕОМ, створення ефекту віртуальної реальності в комп'ютерних іграх і т.д.

Як було наведено вишче, ігрові моделі є однією з груп комп'ютерного моделювання, згрупованих за цілями моделювання. Спочатку теорія ігор використовувалася для опису і моделювання поведінки людських популяцій. Деякі дослідники вважають, що за допомогою визначення рівноваги в відповідних іграх вони можуть передбачити поведінку людських популяцій в ситуації реальної конфронтації. Такий підхід до теорії ігор останнім часом піддається критиці з кількох причин [8].

Припущення, що використовуються при моделюванні, найчастіше порушуються в реальному житті. Дослідники можуть припускати, що гравці обирають поведінки, максимізуючи їх сумарну вигоду (модель економічного людини), однак на практиці людську поведінку часто не відповідає цій передумові. Існує безліч пояснень цього феномена – нераціональність, моделювання обговорення, і навіть різні мотиви гравців (включаючи альтруїзм). Автори теоретико-ігрових моделей заперечують це, кажучи, що їх припущення аналогічні подібним припущенням у фізиці. Тому навіть якщо їх припущення не завжди виконуються, теорія ігор може використовуватися як розумна ідеальна модель, за аналогією з такими ж моделями у фізиці.

Питання про вибір стратегії в ігровому процесі залишається відкритим на даний час. Як зазначалося раніше, комп'ютерне моделювання в контексті теорії ігор може допомогти людям обрати праивльну стратегію в будь-якій справі: військові дії, політика, економіка, спортивні та комп'ютерні ігри і так далі. Взагалі, можна перелічити наступні ознаки гри:

- учасники та їх стратегії;
- логіка поведінки участників;
- неспівпадання інтересів учасників;
- взаємопов'язаність поведінки учасників;
- наявність правил гри, що знають усі учасники.

Дуже корисним міг би бути інструмент, що допоміг би моделювати будь-яку ситуацію, задавати правила гри та логіку учасників. Основною задачею, що необхідно виконати перед розробкою такого інструменту є порівняння способів комп'ютерного моделювання та обрання найкращого з них залежно від цілей моделювання. Пригадаємо основні види комп'ютерного моделювання:

- фізичне моделювання;
- динамічне або чисельне моделювання;
- імітаційне моделювання;
- статистичне моделювання;
- інформаційне моделювання;
- моделювання знань.

Фізичне моделювання допомагає перетворити фізичні сигнали на комп'ютерні і передати їх в існуючу модель з логікою емульованого явища. Цей спосіб моделювання надто залежить від предметної області і не дозволить створити єдиний узагальнений інструмент моделювання ігрової ситуації. Отже, цей спосіб не задовільняє поставленій меті.

Динамічне моделювання добре підходить для систем, де чітко простужуються закони, що можна описати математичними формулами та рівняннями. Програма, що емулюватиме чисельну модель, буде вимагати від користувача вводу математичних рівнянь, що описують систему. Після цього вона повинна буде вивести результат розв'язання рівняння. Така система потребуватиме первинної обробки користувачем, тобто створення системи рівнянь перед внесенням їх у програму. Цей підхід потребує додаткових наукових знань у користувача, отже цільова аудиторія такого додатку значно зменшується. Окрім того, не кожний ігровий процес можна легко завдати за допомогою системи рівнянь. Воне може бути дуже великою і громоздкою, отже потребуватиме великої кількості часу. Плюсом такої моделі є точніть, мінусами є необхідність математичної первинної обробки людиною та складність системи. Програма, що реалізує таку модель повинна лише вирішувати системи рівнянь. Найпоширенішою програмою, що задовільняє таким цілям є Wolfram.

Імітаційне моделювання полягає в описі правил та логіки дій гравців під час ігрового процесу. Отже, користувачу необхідно буде завдати в програмі логіку ігрового процесу за допомогою візуального конструктора. Людина не повинна буде мати знання в додткових сферах. Вона лише матиме знати правила гри та матиме завдати основну логіку дій гравців. Після запуску емуляції процесу можна буде побачити результат вибору тієї чи іншої тактики. Мінусом цього методу є велика громоздкість логіки, а отже великий час на введення даних від користувача. Іншим мінусом є складність анімації логіки гри. Розробити універсальний спосіб описання анімації логіки користувачем є дуже складною задачею, бо існує надто велика кількість дій, що відображається по-різному. Отже, анімація повинна буде вже запрограмована в системі, що накладає обмеження на предметну область використання програми.

Задачею статистичного моделювання є збір і порівняння результатів емуляції в залежності від вхідних даних. Маючи велику базу знань, програми, що реалізують цю модель, можуть повернути вірогідність різноманітних результатів. Використовуючи ці результати можна створити статистику по різним параметрам системи. Головним мінусом цього підходу є потреба в великій базі знань. Також користувач повинен розуміти, які ознаки він може використовувати задля досягнення найточнішого результату. Плюсом є проста робота з програмою, що потребує лише запуску.

Інформаційна модель лише дозволяє за допомогою комп'ютерних засобів наглядно змоделювати предмет чи явище. Вона не містить в собі ніякої логіки, тобто не змінюється з часом. Отже вона не підходить для моделювання ігрових процесів.

Модель моделювання знань схожа на імітаційну модель. Вона також моделює логіку гравців та правила гри. З іншого боку рішення цієї моделі можуть залежти не від запрограмованих правил а від навчання на таких же ситуаціях, що мали місце в минулому. Ця частина моделювання знань схожа на статистичну модель.

Враховуючи аналіз існуючих способів моделювання комп'ютерних систем в контексті створення узагальнюючого конструктора логіки ігор можна виділити дві основні моделі, що необхідно порівняти: імітаційне моделювання та статистичне моделювання.

Закон функціонування має наступний вигляд:

$$Y(t) = F(X, V, H, t), \quad (1)$$

де $X = (x_1, x_2, \dots, x_n)$, – вектор вхідних впливів (стимулів), $Y = (y_1, y_2, \dots, y_m)$ – вектор вихідних сигналів (відгуків, реакцій), $V = (v_1, v_2, \dots, v_k)$ – вектор впливів зовнішнього середовища, $H = (h_1, h_2, \dots, h_l)$ – вектор власних параметрів системи, t – час.

При моделюванні ігрових ситуацій не будемо враховувати вплив зовнішнього середовища. Якщо будуть якісь зошнішні параметри, що впливають на хід гри, будемо вважати їх вхідними параметрами. Задачею роботи є створення узагальненого конструктора ігрових ситуацій, тому система не може мати ніяких специфічних для предметної області параметрів. Окрім того, статистичний спосіб моделювання не залежить від часу. Від залежить лише від вхідних даних та історичних даних.

Отже, для імітаційної моделі формула закону функціонування має наступний вигляд: $Y(t) = F(X, t)$, а для статистичної моделі – $Y = F(X)$, де F – функція, що генерується на основі великої кількості історичних даних.

Задачею даного дослідження є порівняння цих функцій за допомогою програмного забезпечення для моделювання ігрових ситуацій. Основними характеристиками для порівняння є:

- швидкість роботи;
- складність зміни моделі;
- точність моделі.

Статистична модель дозволяє передбачати результат ігрової ситуації на основі великої кількості історичних даних. Отже, в формулі $Y = F(X)$ для

статистичної регресії F – модель, що була згенерована на цих даних. Для кожної ситуації у нас є набір параметрів, що характеризують систему. Позначимо їх вектором $X = (x_1, x_2, \dots, x_n)$, де n – це кількість параметрів моделі. Кожному набору параметрів відповідає значення Y. Позначимо набір історичних даних як $X_h = (X_1, X_2, \dots, X_m)$, а набір значень $Y_h = (Y_1, Y_2, \dots, Y_m)$, де m – кількість наборів даних. Використовуючи ці набори ми можемо побудувати функцію F, що буде максимально близько задовільняти відношенню $Y_i \approx F(X_i)$. Отримавши таку функцію ми зможемо прогнозувати значення моделі для нових вхідних даних. Цей аналіз називається регресійним[9].

Існують різноманітні регресійні методи. Об'єктом дослідження є ігрові ситуації, отже найчастішими результатами є виграш чи програш ігрока. Тобто залежна змінна Y для формули моделі $Y = F(X)$ є категорійною. В таких випадках застосовують логістичну регресію [10]. Логістична регресія має наступний вигляд:

$$F(X) = \frac{1}{1 + e^{-\theta X}}, \tag{2}$$

де $\theta = (\theta_0, \theta_1, \dots, \theta_n)$ – коефіцієнти моделі, $\theta X = \theta_0 + \theta_1 \cdot x_1 + \dots + \theta_n \cdot x_n$.

Тоді значення залежної змінної можно записати наступним чином:

$$Y = \begin{cases} 1, & F(X) \geq 0.5 \\ 0, & F(X) < 0.5 \end{cases} \tag{3}$$

Для отримання функції $F(X)$ необхідно мінімізувати функцію:

$$J(\theta) = \sum_{i=1}^{m} (Y_i \cdot \log F(X_i) + (1 - Y_i) \cdot \log(1 - F(X_i))) \tag{4}$$

Для мінімізації функції необхідно, щоб похідна від цієї функції по кожній з залежних змінних дорівнювала 0, тобто для кожного $j \in [0, n]$ виконуватиметися рівняння $\frac{\partial}{\partial \theta_j} J(\theta) = 0$.

Програмно таке рівняння можна вирішити двома способами. Точне вирішення рівняння має складну формулу та займає багато часу і ресурсів. Тому програмно використовують інший спосіб.

Він полягає в ініціалізації $\theta_0, \theta_1, \dots, \theta_n$ випадковими числами, та задання випадкового числа α. Після цього для кожного θ_j, де $j \in [0, n]$ виконують наступну операцію:

$$\theta_j = \theta_j - \alpha \sum_{i=1}^{m} (F(X_i) - Y_i) \cdot x_j^i, \tag{5}$$

де x_j^i – параметр під номером j з набору X_i.

Ця операція повторюється для кожного θ_j доки кожний з них не буде змінюватися.

Зміна даної моделі полягає в зміні усього набору історичних даних. Тобто для додавання нового параметру моделі необхідно його додати в кожний набір X_i, де $i \in [0, m]$. Після цього модель необхідно заново перерахувати. Тобто для зміни моделі необхідно повністю повторити процес створення моделі. Швидкість та точність можливо проаналізувати тільки після програмного впровадження моделі.

Основною думкою імітаційної моделі є завдання користувачем логіки гри, що дозволяє симулювати ігровий процес та приймати комп'ютером рішення. Найпоширенішим способом симуляції ігрового процесу є підхід ECS (Entity-Component-System) [11].

Під час створення гри в програмі є дуже велика кількість типів сутностей, що мають складну ієрархію. Щоб не дублювати логіку часто за допомогою

успадкування класів спільну логіку виносять у базовий клас. Але дуже часто така логіка може виникати у класах що зовсім не пов'язани один з одним. Саме в таких випадках допомагає ECS. Уся програма поділяється на три частини: сутності, компоненти та системи.

Компоненти – це об'єкти-контейнери, що не володіють властивостями, а виступають сховищами для компонентів. Компоненти – це блоки даних, що визначають всілякі властивості будь-яких ігрових об'єктів або подій. Всі ці дані, згруповані в контейнери, обробляються логікою, яка існує виключно у вигляді систем – чистих класів з певними методами для виконання. Даний патерн є незалежним від будь-якого движка і може бути реалізований багатьма способами. Саме тому цей патерн підходять для розв'язання поставленої задачі. Всі сутності, системи і компоненти повинні десь зберігатися і якимось чином ініціюватися – все це є особливостями реалізації кожного ECS рішення для конкретного движка.

Перевагою статистичного методу моделювання є те, що однаковий підхід може бути використаний для будь-якої предметної області. Користувач повинен обрати характеристики системи, що вважає впливовими для вихідного значення. Після цього він збирає великий датасет, складається з відповідності набору значень характеристик системи та вихідного значення. Користувачу необхідно лише перевести ці характеристики в числовий формат. Недоліком є необхідність в великому датасеті для точного результату. При зміні моделі, наприклад, при додаванні характеристики, необхідно змінити весь датасет і знову навчити модель на новому датасеті.

Для імітаційної моделі нема потреби в датасеті. Необхідно лише скласти дуже детальний набір правил. Точність моделі буде залежати від цих правил поведінки об'єктів. Для зміни моделі необхідно лише додати або видалити правила поведінки. Тобто, зміна імітаційної моделі проходить легше ніж статистичної. Також плюсом імітаційної моделі є відсутність необхідності навчання. Головним недоліком імітаційної моделі є залежність від предметної області. Тобто для зміни правил гри необхідне програмне втручання експерта.

Використана література

1. Гра [Електронний ресурс] / Wikipedia, the free encyclopedia. – Режим доступу: https://uk.wikipedia.org/wiki/%D0%93%D1%80%D0%B0 – 21.10.2019 р. – Назва з екрана.

2. Йоган Гейзинга. Досвід визначення ігрового елемента культури., навч. посіб. – Київ: Основи, 1994. – 20 с.

3. Мороз О. С., Шинкарук В. І.; Л. В. Озадовська. Моделювання – Київ: Абрис, 2002. – С. 392. – 742.

4. Советов Б.Я., Яковлев С.А. Моделювання систем: підр. для вузів – Москва: Висш. Шк., 2001. – 343 с.

5. Боев В.Д., Сипченко Р.П., Комп'ютерне моделювання. – Москва: Інтуіт, 2010. – 349 с.

6. Дворецький С.І., Муромцев Ю.Л., Погонин В.А. Моделювання систем. – Москва: Академія, 2009. – 320 с.

7. Паничев В.В., Солов'їв Н.А. Комп'ютерне моделювання: навч. посіб. – Оренбург: ГОУ ОГУ, 2008. – 130 с.

8. Нейман Дж., Моргенштерн О. Теорія ігор та економічна поведінка – Москва: Наука, 1970.

9. Фрідман Д. Статистичні моделі: теорія та практика. Кембрідж: Видатництво кембріджського університету, 2009. – 128 с.

10. Машинне навчання [Електронний ресурс] / Coursera. – Режим доступу: https://www.coursera.org/learn/machine-learning/home/info – 21.10.2019 р. – Назва з екрана.

11. Entity Systems [Електронний ресурс] / T-machine.org. – Режим доступу: http://t-machine.org/index.php/2007/09/03/entity- systems-are-the-future-of-mmog-develop ment-part-1/ – 21.10.2019 р. – Назва з екрана.