RAG-2

Rzeszów University of Technology Games for Artificial Intelligence 2.0

Tworzenie minigier w standardzie RUT-AI Games 2.0

Marcin Bator 173592

Paweł Buczek 173599

Bartłomiej Krówka 173650

Opiekun: **dr inż. Dawid Kalandyk**

Politechnika Rzeszowska 2025

Tworzenie nowych gier

Wstęp

Zaimplementowane w aplikacji gry są napisane w języku TypeScript. Logika wspólna dla wszystkich gier została wydzielona do klasy `BaseGameComponent`, co pozwala programiście tworzącemu nową grę na skupienie się wyłącznie na logice specyficznej dla gry i ściąga z niego obowiązek dostosowywana logiki komunikacji z innymi mikroserwisami i modelami AI. Do stworzenia strony wizualnej gry można użyć elementu HTML `canvas`, który pozwala na dynamiczne renderowanie obiektów 2D. Jako przykładu stworzenia nowej gry posłuży gra `flappybird`.

Wygenerowanie plików gry

Na początku należy wybrać nazwę gry. Nazwa nie powinna zawierać spacji, pauz, podkreślników - jedynie małe litery alfabetu angielskiego, a także powinna być jak najkrótsza.

W głównym katalogu projektu uruchomić komendę, która uruchomi skrypt tworzący pliki niezbędne do utworzenia gry:

```
npm run newGame flappybird
```

Pliki w których powinna znaleźć się implementacja gry zostaną przez skrypt umieszczone w folderze `src/app/game/games/flappybird`. W tym katalogu znajdują się pliki `flappybird.component.ts` oraz `flappybird.class.ts`. Po uruchomieniu skryptu dostępna stanie się także podstrona o adresie `/game/flappybird`.

Definicja gry

W pliku `flappybird.class.ts` powinny znaleźć się dwie klasy: `FlappyBirdState` dziedzicząca po `TGameState` oraz `FlappyBird` dziedzicząca po `Game`.

Klasa `FlappyBirdState` powinna zawierać właściwości specyficzne dla gry, np. pozycję gracza, prędkość, punkty, itp. Wartości te będą przesyłane do serwisu AI i zapisywane w bazie danych. Należy używać ich podczas implementacji gry. Przykładowa klasa znajduje się na listingu 1.

Listing 1: Klasa FlappyBirdState

```
export class FlappyBirdState implements TGameState {
    public birdY = 0;
    public birdSpeedY = 0;
    public gravity = 0;
    public jumpPowerY = 0;
    public obstacleSpeed = 0;
    public score = 0;
    public difficulty = 0;
    public obstacles = Array.from({ length: 4 }, () => ({
        distanceX: 0,
        centerGapY: 0,
    }));
    public isGameStarted = false;
}
```

Klasa `FlappyBird` powinna nadpisywać pola klasy `Game`, po której dziedziczy. Należy ustawić nazwę gry, zmienną stanu z użyciem klasy `FlappyBirdState`, zmienną tekstową ze specyfikacją danych z klasy stanu oraz tablicę graczy. Przykładowa klasa znajduje się na listingu 2.

Listing 2: Klasa FlappyBird

```
export class FlappyBird extends Game {
    public override name = 'flappybird';
    public override state = new FlappyBirdState();
    public override outputSpec = "
            output:
            birdY: float, <0, 600>;
            birdSpeedY: float, <-inf, inf>;
            gravity: float, <0.5, 1>;
            jumpPowerY: float, <5, 15>;
            obstacleSpeed = <2, 10>;
            score: int, <0, inf>;
            difficulty: int, <0, inf>;
            obstacles: [{distanceX: int, <0, 1900>, centerGapyY: int <100,
                500>}];
            isGameStarted: boolean;

            default values:
            birdY: 300;
            birdSpeedY: 0;
            gravity: 0.5;
            jumpPowerY: 10;
            obstacleSpeed: 2;
            score: 0;
            difficulty: 1;
            isGameStarted: false;
        ";

    public override players = [
        new Player(
        0,
        true,
        'Player 1',
        { jump: 0 },
        {
            ' ': {
            variableName: 'jump',
            pressedValue: 1,
            releasedValue: 0,
            },
        },
        '<jump>:  value of {0, 1}, 0: not jump, 1: jump',
        { jump: '[SPACE]', start: '[SPACE]' }
        ),
    ];
}
```

Tablica `players` jest kluczowym elementem działania rozgrywki i połączenia z modelami zewnętrznymi. Każdy element tablicy to obiekt klasy `Player`, który zawiera informacje o graczu, jego akcjach, opisie akcji, przypisanych klawiszach oraz opisie klawiszy. Pola wymagane przez konstruktor klasy `Player` zostały przedstawiona na listingu 3.

Listing 3: Klasa Player

```
export class Player {
    public id: number;
    public isObligatory: boolean;
    public name: string;
    public inputData: TExchangeData = {};
    public controlsBinding: Record<string, IPlayerControlsBinding> = {};
    public expectedDataDescription = '';
    public controlsDescription: TExchangeData = {};
    public playerType: PlayerSourceType;
}
```

Pole `isObligatory` określa, czy gracz jest obowiązkowy, czy też może być nieaktywny (np. w grze Chińczyk może być od 2 do 4 graczy, więc 2 z nich jest obowiązkowa, a 2 nie).

Pole `inputData` powinno być obiektem typu klucz-wartość, gdzie kluczem jest nazwa parametru wejściowego gracza, zaś wartością: wartość domyślna parametru. W przypadku gry FlappyBird gracz ma tylko jeden parametr wejściowy: ruch. W sterowaniu klawiaturą wartość parametru jest zmieniana przez spację i jest równa 1, gdy gracz naciska spację, a 0, gdy nie naciska. Dostęp do wartości parametru odbywa się poprzez pobranie `inputData['jump']` z odpowiedniego elementu tablicy `players`.

Zachowanie klawiatury jest opisane w polu `controlsBinding`. To również obiekt typu klucz-wartość, gdzie kluczem jest symbol klawisza (w tym przypadku ' ', czyli spacja), zaś wartością obiekt typu `IPlayerControlsBinding`. Zawiera on 3 pola: pole `variableName` określa nazwę parametru wejściowego (zdefiniowanego wyżej), pole `pressedValue` określa wartość parametru, gdy klawisz jest wciśnięty, a pole `releasedValue` określa wartość parametru, gdy klawisz jest zwolniony. Dzięki temu podczas implementacji gry nie trzeba martwić się o obsługę klawiatury, ponadto logika zostanie też prawidłowo obsłużona przez serwis AI.

Ostatnie dwa pola zawierają jedynie informacje dla użytkownika. Pierwsze z nich opisuje, jakie wartości mogą przyjmować parametry wejściowe gry. Jest to istotne z punktu widzenia programisty implementującego serwis integrujący się z grą. Drugie z pól służy do wyświetlania, jakie klawisze są przypisane do akcji.

Implementacja gry

Skrypt generujący komponent stworzył w nim element `canvas`, a także licznik FPS i deklaracje metod potrzebnych do zapisania logiki gry. Przykładowy dekorator klasy komponentu z logiką gry znajduje się na listingu 4.

Listing 4: Dekorator komponentu gry

```
@Component({
    selector: 'app-flappybird',
    standalone: true,
    imports: [CanvasComponent],
    template: `<div>
        score: <b>{{ game.state.score }}</b
        >, difficulty: <b>{{ game.state.difficulty }}</b
        >, jumpPower: <b>{{ game.state.jumpPowerY }}</b
        >, gravity: <b>{{ game.state.gravity }}</b
        >, obstacle speed: <b>{{ game.state.obstacleSpeed }}</b
```

```
        ><b>
            {{
            game.state.jumpPowerY === 15 &&
            game.state.gravity === 1 &&
            game.state.obstacleSpeed === 10
                ? ', MAXIMUM DIFFICULTY HAS BEEN REACHED!'
                : ''
            }}
        </b>
        </div>
        <app-canvas #gameCanvas></app-canvas> <b>FPS: {{ fps }}</b> `,
})
```

Dekorator zawiera ponadto wyświetlanie różnych wartości stanu gry powyżej okienka canvas. Pod nim wyświetla się wartość klatek na sekundę.

Klasa obowiązkowo powinna posiadać pole `game` nadpisujące pole z klasy nadrzędnej, będące typu `FlappyBird`, a więc klasy utworzonej w poprzednim kroku. Rzutowanie typu należy wykonać w metodzie `ngOnInit`, wywołując uprzednio metodę z klasy nadrzędnej.

W klasie mogą znaleźć się też pola niezwracane do websocketu, które zawierają informacje nieistotne dla działania gry, takie jak szerokość gracza lub przeszkody. Jeśli klasa będzie nadpisywać metody OnDestroy lub inne z klasy nadrzędnej, należy odpowiednio uzupełnić nagłówek klasy. Należy przy tym przestrzegać zasad dziedziczenia zawartych w dokumentacji klasy `BaseGameComponent`, np. obowiązkowe wywołanie metody klasy nadrzędnej w metodzie przeładowanej.

Najczęściej w klasie gry powinny zostać przeładowane metody `ngAfterViewInit`, w której zostanie wywołany reset stanu gry i pierwsze wyrenderowanie zawartości, metoda `restart` restartująca stan gry oraz metoda `update` aktualizująca stan gry. We wszystkich tych metodach należy pamiętać o wywołaniu metody z klasy nadrzędnej.

Pozostała część kodu powinna renderować stan gry na elemencie canvas i odpowiednio obsługiwać wydarzenia używając właściwości stanu zdefiniowanych w klasie gry. Nie powinno się implementować własnego przechwytywania zdarzeń klawiatury, ponieważ nie będą one obsługiwać poprawnie komunikacji przez websocket. Przykładowa implementacja głównych metod klasy i wywołanie pozostałych znajduje się na listingu 5.

Listing 5: Klasa FlappyBirdComponent

```
export class FlappyBirdComponent
extends BaseGameWindowComponent
implements OnInit, AfterViewInit, OnDestroy
{
    private _birdWidth = 20;
    private _birdHeight = 20;
    private _obstacleWidth = 50;
    private _obstacleGapHeight = 200;
    private _minDistanceBetweenObstacles = 200;
    private _passedObstacles: boolean[] = new Array(4).fill(false);

    public override game!: FlappyBird;

    public override ngOnInit(): void {
        super.ngOnInit();

        this.game = this.game as FlappyBird;
    }

    public override ngAfterViewInit(): void {
        super.ngAfterViewInit();
```

```
22
23          this.resetBirdAndObstacle();
24          this.render();
25      }
26
27      public override restart(): void {
28          this.game.state = new FlappyBirdState();
29
30          this.game.state.isGameStarted = false;
31          this.resetBirdAndObstacle();
32          this.resetScoreAndDifficulty();
33      }
34
35      protected override update(): void {
36          super.update();
37
38          if (
39          !this.game.state.isGameStarted &&
40          this.game.players[0].inputData['jump'] === 1
41          ) {
42          this.game.state.isGameStarted = true;
43          }
44
45          if (!this.isPaused && this.game.state.isGameStarted) {
46          this.updateBirdPosition();
47          this.updateObstaclePosition();
48          this.updateBirdSpeed();
49          this.checkCollision();
50          }
51          this.render();
52      }
53  }
```

Podsumowanie

Po zaimplementowaniu gry należy ją przetestować, uruchamiając serwer deweloperski. W przypadku błędów należy sprawdzić, czy wszystkie metody zostały nadpisane, czy wszystkie zmienne są poprawnie zadeklarowane, a także czy nie ma błędów w logice gry. Można wzorować się na już zaimplementowanych grach.