Grafická aplikace v JavaFX

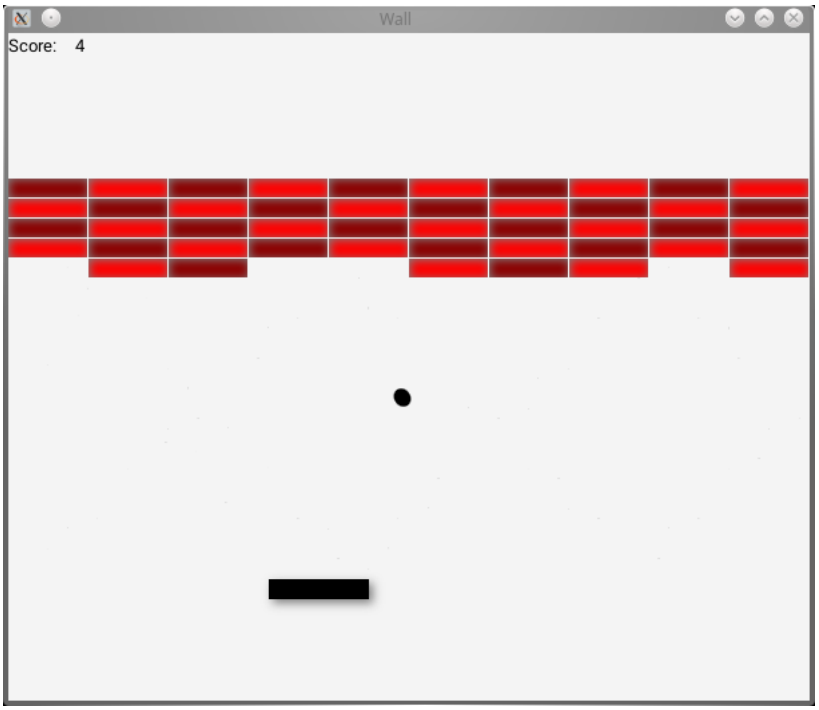
9

Platforma JavaFX vznikala jako přímá konkurence platformy Adobe Flash, z čehož plyne kvalitní podpora pro práci (nejen) s vektorovou grafikou, tvorbu animací nebo interaktivních aplikací. Část těchto schopností si ukážeme v tomto semináři na jednoduchém remaku klasické osmibitové hry Arkanoid[1], v českých zemích známé též jako *bourání zdi*.

1 Popis hry

Pro ty, co se s touto hrou ještě nesetkali, připomeňme její podstatu. Ústředním objektem této hry je míček (kulička) pohybující se diagonálně po hrací ploše. V horní části herní plochy je několik řad cihel, které míček při nárazu rozbíjí. V dolní části herní plochy je hráč, resp. jeho pálka, který může míček odrazit. Pokud míček narazí do cihly, pálky, horního, levého nebo pravého okraje, je odražen. Pokud míček dopadne na spodní stranu herní plochy, hra končí.

Jak by měla hra nakonec vypadat ukazuje Obrázek 1.



Obrázek 1: Vzhled zamýšlené hry

[1] `https://en.wikipedia.org/wiki/Arkanoid`

2 Protyp: Pohybující se kolečko

Začneme tím, že si vytvoříme míček, který rozpohybujeme. Vytvoříme si standardní JavaFX aplikaci, jako v předchozích seminářích.

```
public class WallApp extends Application {
    private Pane gamePane;

    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) throws Exception {
        Ellipse ballShape = new Ellipse(20, 20);
        ballShape.setCenterX(200);
        ballShape.setCenterY(50);

        Pane gamePane = new Pane();
        gamePane.getChildren().add(ballShape);

        Scene scene = new Scene(gamePane, 600, 500);
        primaryStage.setTitle("Wall");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

Nejdříve jsme vytvořili elipsu (`ballShape`) o šířce a výšce 20×20 (tedy kolečko) a určili jsme souřadnice jeho středu (200, 50). Dále jsme vytvořili jednoduchou plochu typu `Pane`, kam jsme kolečko umístili.

K rozpohybování kolečka použijeme objekt typu `Timeline`, který (mimo jiné) v definovaných časových intervalech umožňuje vyvolat událost. Následující kód je vhodné umístit před vytvoření scény.

```
Timeline timeline = new Timeline();
KeyFrame updates = new KeyFrame(
    Duration.millis(100),
    e -> {
        ballShape.setCenterX(ballShape.getCenterX() + 5);
    });
timeline.getKeyFrames().add(updates);
timeline.setCycleCount(200);
timeline.play();
```

Třída `Timeline` je primárně určena pro práci s animacemi, a proto pro klíčové události, které se ve scéně

objeví, používá označení *klíčové políčko*, resp. `KeyFrame`[2]. V tomto případě jsme vytvořili `KeyFrame`, který se opakuje v 100 ms intervalech. Každých 100 ms je vyvolána událost, která posune kolečko o 5 pixelů vpravo. Dále jsme stanovili, že tato událost má být vyvolána 200×. Pokud bychom chtěli nějakou událost opakovat donekonečna, použili bychom místo 200 konstantu `Animation.INDEFINITE`. Závěrem animaci spustíme zavoláním metody `timeline.play()`.

Kolečko se sice hýbe na jednu stranu, ale bylo by vhodné zapracovat na „fyzikálním modelu" a přidat mu schopnost odrazu. Pro úsporu místa si ukážeme pouze odraz od vertikálních překážek.

Nejdříve si zavedeme atribut, který bude určovat vektor pohybu kolečka.

```
private int ballDirX = 5;
```

Dále upravíme událost, která zajišťuje pohyb kolečka:

```
double x = ballShape.getCenterX();
if ((x <= 0) || (x >= 300)) ballDirX *= -1;
ballShape.setCenterX(x + ballDirX);
```

Tato událost zjistí souřadnici kolečka a ověří, zda již nebylo dosaženo hranice, kam se může pohybovat. Pokud ano, je vektor pohybu otočen. V každém případě se kolečko posune ve směru daného vektoru.

Nyní jsme dostali náš program do stavu, kdy pomalu dělá, co od něj potřebujeme. Zároveň je to moment, kdy je potřeba říct: „**DOST! Takto se to opravdu nedá dělat!**"

Program trpí celou řadou problémů:

(i) V první řadě třída `WallApp` porušuje princip jedné odpovědnosti[3]. Tato třída obstarává uživatelské rozhraní (to by měla být její hlavní úloha), ale vedle toho řeší logiku aplikace (odrážení míčku) nebo drží směr, kterým se míček pohybuje.

(ii) Co je míček, je určeno vykreslovaným objektem. Další jeho vlastnosti, např. vektor pohybu musí být umístěny v oddělené třídě. Informace vztahující se k míčku, by měly být na jednom místě.

(iii) Kód obsahuje obrovské množství *magických konstant*, číselných hodnot, jejichž význam je v lepším případě znám alespoň autorovi aplikace.

V této fázi jde programu ještě rozumět, ale pokud bychom pokračovali i nadále v tomto duchu, patrně bychom hru byli schopni dotáhnout do konce, ale bylo by to za cenu naprosto nesrozumitelného a neudržovatelného kódu.

Sluší se dodat, že nejeden program začíná jako prototyp, který má tyto neduhy. Je však nutné být schopen tyto problémy identifikovat a program zrefaktorovat tak, aby jimi netrpěl. Proto v následujících kapitolách

(i) rozdělíme kód do samostatných tříd se svou vlastní zodpovědností (uživatelské rozhraní, míček, stav hry),

(ii) zavedeme model hry nezávislý na grafickém rozhraní,

(iii) budeme používat konstanty tak, aby na jednom místě bylo možné měnit vlastnosti hry.

[2] *Key frame* obvykle značuje políčko animace, kde se scéna nějak zásadně mění.

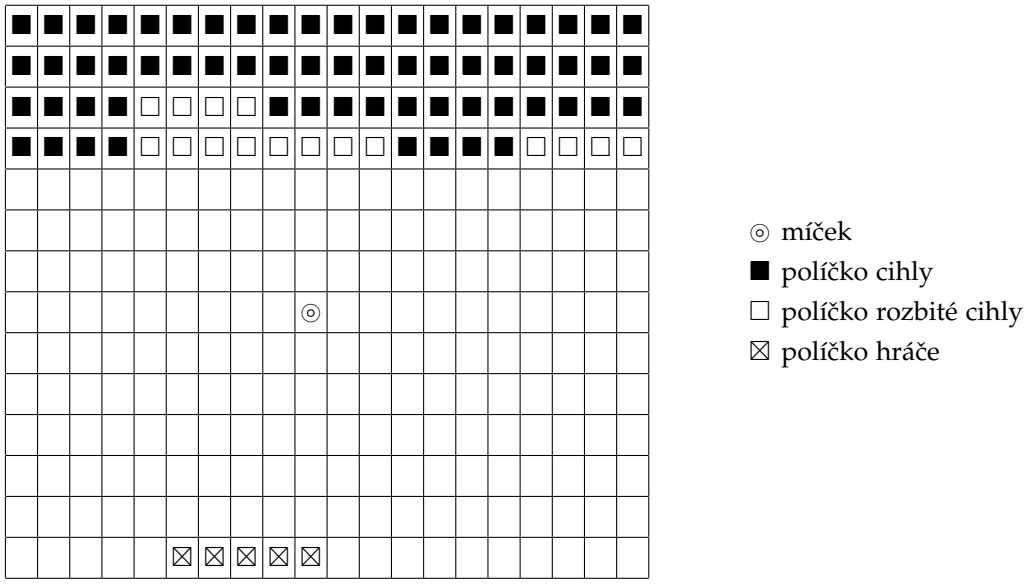
[3] Single responsibility principle `https://en.wikipedia.org/wiki/Single-responsibility_principle`

3 Implementace hry

Nejdříve si představíme model hry (logiku hry), jeho realizaci, a poté jej propojíme s uživatelským rozhraním.

3.1 Model hry

Hry typu Arkanoid vznikly pro velice primitivní osmibitové počítače, které ani v nejmenším neumožňovaly na pixel přesnou grafiku. Proto používají zjednodušený model, kdy je plocha hry rozdělena do menších políček a každé políčko může zabírat nanejvýš jeden objekt jednoho typu, např. míček, cihla, pálka, jak ilustruje Obrázek 2. Tento model budeme používat také, protože umožňuje velice pohodlně řešit interakce (kolize) mezi objekty.



Obrázek 2: Ilustrace vnitřní reprezentace hry

Hru rozdělíme do čtyř samostatných tříd:

1. `WallApp` – obstarává uživatelské rozhraní,
2. `GameState` – drží stav hry (objekty míčku, cihel, pálky, score, atp.),
3. `Ball` – reprezentuje pozici a směr pohybu míčku,
4. `Brick` – reprezentuje cihlu a kromě její souřadnice má příznak, zda je rozbitá, nebo ne.

3.2 Míček aneb pohybující se kolečko 2.0

Vytvoření pohybujícího se míčku si rozdělíme do dvou fází: (i) reprezentace míčku a (ii) jeho zobrazení v aplikaci.

3.2.1 Reprezentace míčku

Vytvoříme třídu `Ball`, která bude mít čtyři atributy `row`, `column`, `rowDirection` a `colDirection`, které určují polohu míčku (řádek a sloupec) a směr jeho pohybu (ve směru řádků a sloupců). Samotný posun míčku bude řešit metoda `void update()`.

Dále potřebujeme mechanizmus, kterým objekt bude poskytovat informaci o své pozici. Určitým řešením by bylo vytvořit[4] odpovídající gettery a settery a v třídě `WallApp` na základě těchto hodnot nastavit souřadnice kolečka. Avšak JavaFX pro tyto situace nabízí výrazně pohodlnější řešení, kterým je systém *vlastností*.

Systém *vlastností* (properties) v JavaFX povyšuje atributy objektů na plnohodnotné objekty. Díky tomu je možné sledovat změny těchto vlastností[5], provázat vlastnost jednoho objektu s vlastností jiného objektu nebo provádět postupné transformace vlastností. To v konečném důsledku umožňuje oddělit aplikační a prezentační logiku programu.

Základní kostra třídy `Ball` by mohla vypadat následovně:

```
public class Ball {
    private IntegerProperty row;
    private IntegerProperty column;
    private int rowDirection = -1;
    private int colDirection = -1;
    public Ball(int r, int c) {
        row = new SimpleIntegerProperty(r);
        column = new SimpleIntegerProperty(c);
    }
}
```

Atributy `row` a `column` jsme vytvořili jako vlastnosti typu `IntegerProperty` a v konstruktoru je inicializovali objekty typu `SimpleIntegerProperty`, které implementují vlastnost (property) obsahující celočíselnou hodnotu (podobně máme `SimpleDoubleProperty` nebo `SimpleBooleanProperty`). Díky tomu bude možné provázat souřadnice míčku (třída `Ball`) s kolečkem, které bude představovat vykreslený míček (třída `Ellipse`). Aby toto provázání bylo možné uskutečnit, je nutné vytvořit[6] odpovídající JavaFX gettery a settery. Ty mají trochu jinou strukturu něž klasické gettery a settery u POJO, jak ukazuje příklad getterů/setter pro vlastnost `row`.

```
public final IntegerProperty rowProperty() {
    return this.row;
}
public final int getRow() {
    return this.rowProperty().get();
}
public final void setRow(final int row) {
```

[4] Nechat si vygenerovat vývojovým prostředím.
[5] Návrhový vzor Observer
[6] Nechat si vygenerovat vývojovým prostředím.

```
    this.rowProperty().set(row);
}
```

První getter zpřístupňuje samotným objekt vlastnosti, zbývající dva imitují klasické gettery a settery, tj. zpřístupňují hodnotu skrytou uvnitř objektu vlastnosti.

Atributy `rowDirection` a `colDirection` vytvoříme jako tradiční soukromé atributy typu `int`, protože ty slouží k implementaci chování míčku (jeho pohybu a odrazu) a není nutné, aby k nim někdo přistupoval z vně objektu.

Kód metody, která posune míček je přímočarý, viz:

```
public void update() {
    setRow(getRow() + rowDirection);
    setColumn(getColumn() + colDirection);
}
```

Aby byl kód třídy `Ball` úplný, musíme ještě implementovat metody pro odraz od překážek:

```
/** Odrazi mic od vodorovne prekazky */
public void horizontalBounce() {
    rowDirection *= -1;
}
/** Odrazi mic od vertikalni prekazky */
public void verticalBounce() {
    colDirection *= -1;
}
```

U těchto metod aplikujeme stejný přístup, který jsme si ukázali v Kapitole 2.

3.2.2 Zobrazení a pohyb míčku

Pro vyzkoušení můžeme s několika málo rozdíly využít kód z Kapitoly 2.

Vytvoříme objekt míčku na nějakých souřadnicích:

```
Ball ball = new Ball(10, 20);
```

Vytvoříme objekt kolečka:

```
Ellipse ballShape = new Ellipse(20, 20);
```

Avšak nebudeme mu nastavovat konkrétní souřadnice, ale následovně je provážeme s vlastnostmi objektu `ball`:

```
ballShape.centerXProperty().bind(ball.columnProperty());
ballShape.centerYProperty().bind(ball.rowProperty());
```

Díky použití metody `centerXProperty().bind(IntegerProperty)` a `centerYProperty().bind(IntegerProperty)` budou souřadnice kolečka určeny řádkem a sloupcem míčku. Kdykoliv změníme sloupec nebo řádek u objektu `ball`, promítne se do vykresleného kolečka.

Můžeme si to vyzkoušet tak, že do `timeline` vložíme volání metody `ball.update()`.

Po spuštění se míček sice posunuje, ale pouze po jednotlivých pixelech. Je to z toho důvodů, že zobrazení odpovídá 1:1 modelu, se kterým pracujeme. Což není žádoucí. Naším cílem je, aby jedno políčko modelu odpovídalo několika pixelům v zobrazení hry.

Zavedeme proto konstantu, která bude udávat, kolika pixelům odpovídá jedna jednotka v modelu hry.

```
private static final int UNIT_SIZE = 15;
```

Kolečko přizpůsobíme této konstantě:

```
Ellipse ballShape = new Ellipse(UNIT_SIZE / 2, UNIT_SIZE / 2);
```

Dále musíme upravit provázání vlastností, aby reflektovaly tuto konstantu. Právě pro tyto účely je možné z vlastností odvozovat vlastnosti nové. Máme-li například číselnou vlastnost, můžeme vytvořit vlastnost odvozenou, která bude jejím násobkem nebo která bude o nějaké číslo k větší, či menší.

Upravíme proto provázaní objektu `ballShape` s objektem `ball`, aby souřadnice kolečka nebyly určeny přímo sloupcem a řádkem, ale vlastností, které ja násobkem sloupce a řádku:

```
ballShape.centerXProperty().bind(ball.columnProperty().multiply(UNIT_SIZE));
ballShape.centerYProperty().bind(ball.rowProperty().multiply(UNIT_SIZE));
```

Nyní jde vidět, že se míček pohybuje po celých políčcích, ale je tu ještě jeden problém, který není zcela vyřešen. Pokud umístíme míček na souřadnice (0, 0), bude z něj vidět pouze jedna čtvrtina. Je potřeba jej o půl políčka posunout. Jinými slovy k souřadnicím středu kolečka je nutné přičíst velikost půlky políčka.

```
ballShape.centerXProperty().bind(ball.columnProperty().multiply(UNIT_SIZE).add(UNIT_SIZE / 2));
ballShape.centerYProperty().bind(ball.rowProperty().multiply(UNIT_SIZE).add(UNIT_SIZE / 2));
```

Nyní už se míček pohybuje správně, ale nezvládá odraz, ten je potřeba řešit v logice hry, které se bude věnovat v následující kapitole.

3.3 Logika hry

V minulých kapitolách jsme si ukázali, jak se dá oddělit logika (chování) míčku od jeho prezentace (zobrazeného kolečka). Podobný princip aplikujeme i na celou hru. Vše zabalíme do objektu jedné třídy, který bude obsahovat nejen míček, ale i rozbíjené cihly a hráčovu pálku.

3.3.1 Cihla

Než můžeme cihly začít vytvářet a rozbíjet, musíme si pro ně definovat třídu. Ta je jednoduchá, protože cihla nepotřebuje mít žadnou logiku, myšleno herní logiku. U každé cihly budeme evidovat její pozici a to, zda byla rozbita, nebo ne.

Základní struktura třídy, která je nastíněna níže, obsahuje pouze předpokládané vlastnosti. Odpovídající JavaFX gettery a settery jsou pro stručnost z tohoto textu vypuštěny.

```
public class Brick {
    private final BooleanProperty active;
        private final IntegerProperty row;
        private final IntegerProperty column;
        public Brick(int row, int column) {
                this.row    = new SimpleIntegerProperty(row);
                this.column = new SimpleIntegerProperty(column);
                this.active = new SimpleBooleanProperty(true);
        }
}
```

3.3.2 Stav hry

Logiku hry zabalíme do třídy `GameState`, která v první verzi bude obsahovat jen míček (třídá `Ball`) a seznam s objekty cihel (seznam `List<Brick>`). S těmi budeme pracovat jako s bežnými atributy a vytvoříme k nim běžné gettery (pro stručnost nejsou uvedeny).

```
public class GameState {
        private final List<Brick> bricks;
        private final Ball ball;
}
```

Dále si nadefinuje několik pomocných konstant, která udávají počet sloupců cihel, počet řádků s cihlami, šířku cihly, počet řádků herní plochy a počet sloupců. Cílem je, aby program byl závislý pouze na těchto konstantách, a my mohli pomocí jejich nastavení definovat odpovídající vlastnosti hry. Zvolené hodnoty ukazuje následující výřez z kódu.

```
/** pocet sloupcu cihel */
public static final int BRICK_COLS = 10;
/** pocet rad cihel */
public static final int BRICK_ROWS = 5;
/** sirka cihly */
public static final int BRICK_WIDTH = 4;
/** pocet radku herni plochy vcetne cihel */
public static final int ROWS = 20;
/** pocet sloupcu herni plochy */
public static final int COLUMNS = BRICK_COLS * BRICK_WIDTH;
```

Míček i cihly vytvoříme v konstruktoru, jak ukazuje následující kód.

```
public GameState() {
    this.ball      = new Ball(ROWS / 2, COLUMNS / 2);
```

```
    this.bricks    = new ArrayList<>();
    for (int r = 0; r < BRICK_ROWS; r++)
        for (int c = 0; c < BRICK_COLS; c++)
            bricks.add(new Brick(r, c));
}
```

Míček je na počátku umístěn do středu hřiště a jsou vytvořeny jednotlivé řady cihel. Všimněme si, že v tento okamžik se nikde neuvažuje šířka cihly, ale jen jejich vzájemné uspořádání.

Součástí stavu hry by měla být i metoda, která jej bude v pravidelných intervalech aktualizovat. My si ji pojmenujeme `void update()`. V její první verzi se bude odrážet míček od všech stěn a rozbíjet cihly, se kterými se potkal. Její kód by mohl vypadat následovně.

```
public void update() {
    ball.update();

    int r = ball.getRow();
    int c = ball.getColumn();

    if ((c == 0) || (c == COLUMNS - 1)) ball.verticalBounce();

    Optional<Brick> collision = bricks.stream()
            .filter(b -> b.getColumn() == c / BRICK_WIDTH)
            .filter(b -> b.getRow() == r)
            .filter(b -> b.isActive())
            .findFirst();
    if (collision.isPresent()) {
        ball.horizontalBounce();
        collision.get().setActive(false);
    } else {
        if ((r == 0) || (r == ROWS)) ball.horizontalBounce();
    }
}
```

Nejdříve dojde k posunu míčku na novou pozici. Poté si pro větší pohodlí souřadnice míčku převedeme z JavaFX vlastností na běžné primitivní typy a budeme řešit kolize. Pokud míček narazí do některé z bočních stěn herní plochy, použijeme odražení, které jsme tu již měli.

Z vertikálního pohledu je situace komplikovanější, protože se může stát, že jsme narazili do cihly. Pro tento případ si vybereme cihly, které jsou v daném sloupci a na daném řádku jako míček a dosud nebyly rozbity. Pro elegantnější zápis je použito Stream API.

Pokud taková cihla existuje, znamená to, že došlo ke kolizi. Proto míček odrazíme a cihlu jež byla s míčkem v kolizi, deaktivujeme. Pokud nedošlo ke kolizi, ověříme, zda míček nenarazil do horního nebo spodního okraje herní plochy a případně jej odrazíme.

Zbývá propojit stav hry a její prezentační stránku. Pro každou cihlu vytvoříme obdélník a vložíme jej na plochu, jak ukazuje následující kód.

```
for (Brick brick : gameState.getBricks()) {
    Rectangle brickShape = new Rectangle(UNIT_SIZE * GameState.BRICK_WIDTH - 1, UNIT_SIZE - 1);

    brickShape.xProperty().bind(brick.columnProperty()
        .multiply(UNIT_SIZE * GameState.BRICK_WIDTH));
    brickShape.yProperty().bind(brick.rowProperty()
        .multiply(UNIT_SIZE));
    brickShape.visibleProperty().bind(brick.activeProperty());

    gamePane.getChildren().add(brickShape);
}
```

Tento kód je velice podobný tomu, co jsme již viděli u míčku. Za zmínku stojí, že cihly jsme udělali o jeden pixel menší než je plný rozměr, aby mezi nima byla vidět mezera. Dále jsme využili propojení mezi vlastností `Rectangle.visiblePropery` a `Brick.activeProperty`, které nám zajišťuje, že kdykoliv v rámci aktualizace hry nějakou cihlu rozbijeme, automaticky se skryje i v uživatelském rozhraní.

Závěrem této podkapitoly podotkněme, že je potřeba aktualizovat objekt `timeline` tak, aby volal `GameState.update()` a nastavit počet cyklů na `Animation.INDEFINITE`.

3.4 Hráč

Aby hra byla kompletní, je potřeba do ní vložit samotného hráče. Nabízelo by se vytvořit pro hráče samostatnou třídu, my jsme se jej pro jednoduchost rozhodli umístit přímo do stavu hry.

Zavedeme si nejdříve pomocné konstanty, které budou definovat, na kterém řádku se pálka nachází a jak je široká.

```
/** velikost hrace (palky) */
public static final int PADDLE_WIDTH =  5;
/** pozice palky */
public static final int PADDLE_ROW = 20;
```

Dále vytvoříme JavaFX vlastnost udávající hráčovu pozici:

```
/** pozice hrace, resp. sloupec, na kterem se nachazi zacatek hrace (palky) */
private final IntegerProperty paddlePos;
```

V konstruktoru ji umístíme na střed.

```
this.paddlePos = new SimpleIntegerProperty((COLUMNS - PADDLE_WIDTH) / 2);
```

Pro ovládání hráče zavedeme dvě metody s vcelku přímočarým kódem, které zajišťují posun hráče a také, aby hráč nevypadl z horní plochy, viz:

```
/** Posune hrace doleva */
public void moveLeft() {
    int paddleCol = paddlePos.getValue();
    if (paddleCol > 0) paddlePos.setValue(paddleCol - 1);
}

/** Posune hrace doprava */
public void moveRight() {
    int paddleCol = paddlePos.getValue();
    if (paddleCol + PADDLE_WIDTH < COLUMNS) paddlePos.setValue(paddleCol + 1);
}
```

Přidání pálky nám taktéž změní logiku hry. Pokud míček dosáhne spodního okraje, musíme rozlišovat, zda se před ním nachází hráčova pálka a měl by být odražen, nebo došlo k propadnutí míčku a k ukončení hry. S tím souvisí, že musíme být schopni signalizovat konec hry. Proto metodu `void update()` změníme na `boolean update()`, a pokud nebude možné pokračovat ve hře, vrátí metoda `false`, jinak `true`.

Původní řádek, jenž se stará o odraz od horizontálních překážek, rozdělíme následovně.

```
if (r == 0) ball.horizontalBounce();
int paddleCol = paddlePos.getValue();

if ((r == PADDLE_ROW - 1) && (c >= paddleCol) && (c < paddleCol + PADDLE_WIDTH))
    ball.horizontalBounce();

if (r == ROWS) return false;
```

Pokud narazíme na horní okraj, míček odrazíme. Dále otestujeme, zda se míček nachází před pálkou. Pokud ano, míček odrazíme. Pokud se, ale míček dostal za hranici herní plochy, hra končí.

Závěrem zbývá udělat propojení s herní plochou. Pro pálku použijeme obdélník ve stejném duchu, jako pro cihly, proto jej tady nebudeme podrobněji rozepisovat.

Důležité ale bude zapojení ovládání z klávesnice. To se odehrává ve vlastnostech scény a my jej převedeme do samostatné metody následovně.

```
scene.setOnKeyPressed(this::dispatchKeyEvents);
// ...
private void dispatchKeyEvents(KeyEvent e) {
    switch (e.getCode()) {
    case LEFT: gameState.moveLeft(); break;
    case RIGHT: gameState.moveRight(); break;
    default:
}
```

Poslední změna, kterou je potřeba dodělat, je zastavení hry, když míček propadne. Toho dosáhneme další

změnou v `timeline`. Jednoduše, pokud metoda `GameState.update()` vrátí `false`, `timeline` (resp. hru) zastavíme, tj. provedeme.

```
if (!gameState.update()) timeline.stop();
```

Nyní jsme vytvořili základ hry, který je sice již použitelný, ale je potřeba jej dále rozvíjet. Přiložené zdrojové kódy řeší další funkcionalitu, jako je opakované spuštění hry, korektní zastavení, zobrazení score, atp., které ale jen opakují principy, jež byly již probrány ať už v tomto semináři nebo jež patří mezi běžné programátorské dovednosti. Proto p.t. čtenáře odkazujeme k nahlédnutí do zdrojových kódů pro další podrobnosti.

4 Vychytávky

Závěrem několik málo vychytávek, které jsou v ukázkové hře použity.

4.1 Efekty

Grafickým objektům (včetně uživatelských prvků jako jsou tlačítka) je možné přiřadit různé grafické efekty, např. stín, rozmazání.

My jsme je použili na dvou místech. V metodě `WallApp.createPaddle()` je vytvořen stín pod hráčem a v metodě `WallApp.createBrick(Brick)` pomocí vnitřního stínu vytváříme plastičtější dojem cihly.

4.2 Třída StackPane

Pro uspořádání jednotlivých prvků jsme použili třídu `StackPane`, která umožňuje skládat objekty do vrstev nad sebou, a proto se často používá v grafických aplikacích.

4.3 Timeline a animace

Třída `Timeline` vedle vyvolání událostí umožňuje definovat průběžnou změnu vlastností. Všimněme si, že v přiloženém kódu je v metodě `WallApp.createBall(Ball)` míček vytvořen mírně šišatě. A dále je k němu vytvořen další objekt třídy `Timeline`.

```
Timeline ballTimeline = new Timeline();
ballTimeline.getKeyFrames().add(
    new KeyFrame(Duration.millis(500),
                 new KeyValue(ballShape.rotateProperty(), 360)));
ballTimeline.setCycleCount(Timeline.INDEFINITE);
ballTimeline.play();
```

V tomto případě klíčový rámec udává, že v průběhu 500 ms se má vlastnost `rotateProperty` změnit z původní hodnoty (což je 0) na 360.

Můžeme tak změnit i další vlastnosti, např. zvětšení:

```
ballTimeline.getKeyFrames().add(new KeyFrame(Duration.millis(1000),
    new KeyValue(ballShape.scaleXProperty(), 2)));
```

Tento kód každou jednu sekundu roztáhne míček na dvojnásobnou šířku a pak jej v jednom okamžiku zase uvede do původního stavu („vyfoukne"). Pokud bychom chtěli hodnotu změnit plynule zpět, můžeme objektu `Timeline` nastavit vlastnost `autoReverse`.

Úkol: Vyzkoušejte si, že to funguje i s dalšími objekty, např. tlačítky.