UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Logická 2D hra v Javě

Miroslav Štolfa

Bakalářská práce
2013

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2012/2013

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Miroslav Štolfa**
Osobní číslo:	**I10234**
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Logická 2D hra v Javě**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování:

Anotace:
Cílem práce je vytvoření logické 2D hry s využitím fyzického enginu JBox2D.
Teoretická část:
V teoretické části budou popsány vhodné technologie pro vývoj herních aplikací. Bude provedena základní analýza vývoje 2D her pro osobní počítače. Dále bude provedena analýza základní herní logiky a budou popsány principy různých druhů herních smyček.
Implementační část:
V implementační části práce budou popsány základní principy a chování hry samotné. Dále bude popsáno využití knihoven JBox2D (fyzika) a Slick2D (vykreslování) a bude realizována ukázková aplikace.

Rozsah grafických prací:

Rozsah pracovní zprávy:

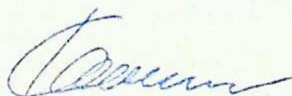
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

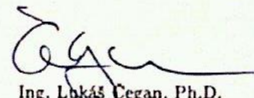
DAWISON, Andrew. Programování dokonalých her v Javě. Vyd. 1. Překlad Lukáš Krejčí. Brno: Computer Press, 2006, 902 s. ISBN 80-722-6944-5.
MORRISON, Michael. Naučte se programovat počítačovou hru za 24 hodin. Vyd. 1. Brno: Computer Press, 2004, 421 s. ISBN 80-251-0371-4.
DARWIN, Ian F. Java: kuchařka programátora. Vyd. 1. Brno: Computer Press, 2006, 798 s. ISBN 80-251-0944-5.

Vedoucí bakalářské práce:	**Ing. Zdeněk Šilar**
	Katedra informačních technologií

Datum zadání bakalářské práce:	**21. prosince 2012**
Termín odevzdání bakalářské práce:	**10. května 2013**



L.S.



prof. Ing. Simeon Karamazov, Dr.
děkan

Ing. Lukáš Čegan, Ph.D.
vedoucí katedry

V Pardubicích dne 29. března 2013

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 9. 5. 2013 Miroslav Štolfa

Poděkování

Chtěl bych poděkovat Lukáši Jirkovskému za pomoc při tvorbě herní grafiky a Ing. Zdeňku Šilarovi za pomoc a podporu při tvorbě bakalářské práce. Dále bych rád poděkoval svojí přítelkyni Tereze Bariové za pevné nervy a stálou podporu. Nakonec bych chtěl poděkovat svojí rodině za vše, co pro mne kdy udělala.

Anotace

Cílem práce je vytvoření logické 2D hry s využitím fyzického modelu JBox2D. V teoretické části jsou popsány základní technologie, které poskytuje Java a základní funkcionalita knihoven Slick2D a JBox2D, které byly využity pro vývoj hry. V implementační části práce jsou popsány základní principy a chování hry samotné. Dále je zde popsáno využití externích knihoven.

Klíčová slova

logická hra, Java, 2D grafika, fyzikální model

Title

Logical 2D game in Java

Annotation

The aim is to create a logical 2D game using physical model JBox2D. The theoretical part describes the basic technology that provides Java libraries and basic functionality Slick2D and JBox2D that were used to develop game. In the implementation part describes the basic principles and behavior of the game itself. It is described the use of external libraries.

Keywords

Logical game, Java, 2D graphics, physics engine

Obsah

Seznam zkratek **8**

Seznam obrázků **9**

Seznam tabulek **9**

Úvod **10**

1 Programovací jazyk Java **11**

1.1 Rozhraní 11

1.2 Kolekce 12

1.3 Výčtový typ 13

2 Herní vývojový nástroj **15**

2.1 Knihovna Slick2D v prostředí NetBeans IDE 15

2.2 Herní cyklus 15

2.3 Grafika 16

2.4 Vstupy a výstupy 16

2.5 Zvuky a hudba 17

2.6 Podpora fontů 18

2.7 Stavově koncipovaná hra 19

2.8 Knihovna Lightweight Java Game Library 19

2.9 Knihovny Jorbis a JOGG 19

3 Fyzikální model hry **20**

3.1 Základní koncept 20

3.2 Moduly 21

3.3 Jednotky 22

3.4 Továrny a definice 22

3.5 Uživatelská data 22

3.6 Simulátor fyzikálního modelu 23

3.7 Pevné těleso a jeho vlastnosti 23

3.8 Typy spojení 24

3.9 Knihovna Fizzy 25

4 Implementace logické 2D hry **26**

4.1 Adresářová struktura hry 26

4.2 Rozdělení hry na stavy 26

4.2.1 Inicializace herních dat ... 28
4.3 Řízení hry ... 29
4.4 Třídy herních prvků ... 29
4.4.1 Kontejnery ... 30
4.4.2 Továrny ... 30
4.4.3 Obecné ... 31
4.4.4 Průhledový displej ... 33
4.4.5 Další nástroje ... 33
4.5 Herní objekty ... 34
4.5.1 Třídy StaticObject a DynamicObject ... 34
4.5.2 Třída ItemObject ... 35
4.6 Hráč ... 35
4.6.1 Robot Isaac ... 35
4.6.2 Inventář ... 41
4.6.3 Úkoly ... 42
4.7 Herní úroveň ... 43
4.8 Efekty ... 44
5 Ovládání aplikace ... 45
5.1 Spuštění aplikace ... 45
5.2 Klávesnice a myš ... 45
Závěr ... 47
Literatura ... 48
Příloha A – Použití JBox2D ... 49

Seznam zkratek

JVM	Java Virtual Machine
LWJGL	Lightweight Java Game Library
OGG	Datový formát, kontejner
WAV	Waveform Audio File Format
PNG	Portable Network Graphics
JPEG	Joint Photographic Experts Group
TGA	Souborový formát Targa
GIF	Graphics Interchange Format
OpenGL	Open Graphic Library
OpenAL	Open Audio Library
TTF	TrueTypeFont
BSD	Berkeley Software Distribution
zlib	Typ licence svobodného softwaru
NPC	Non-Player Character
HUD	Head-up Display

Seznam obrázků

Obrázek 1 – Časové tunelování 21
Obrázek 2 – Demonstrace knihovny JBox2D 23
Obrázek 3 – Průběh načítání hry 27
Obrázek 4 – Úvodní obrazovka s herním menu 28
Obrázek 5 – Průhledový displej ve hře 33
Obrázek 6 – První koncept robota 36
Obrázek 7 – Manipulace s robotickým ramenem 39
Obrázek 8 – Grafická podoba inventáře 42
Obrázek 9 – Návod na skryté obrazovce ve hře 43

Seznam tabulek

Tabulka 1 – Funkční klávesy 45
Tabulka 2 – Funkční tlačítka myši 46

Úvod

Hra, ať už stolní nebo počítačová, by měla být pro hráče určitou výzvou. Lidé jsou od pradávna soutěživí a mají rádi výzvy. Proto jsou hry, jakéhokoliv druhu, mezi lidmi tak oblíbené.

Tato práce je věnována vývoji logické hry v programovacím jazyce Java, s využitím dvou externích knihoven. Základ celé herní aplikace je vytvořen s využitím knihovny Slick2D, která zajišťuje inicializaci, vykreslování a aktualizaci hry. Dále je částečně použita knihovna Lightweight Java Game Library, která pracuje s nativními knihovnami OpenGL a OpenAL. Knihovna Slick2D poskytuje základní nástroje pro tvorbu hry – třídy pracující s obrázky, animací, zvuky a vstupy.

Ve hře je využit fyzikální model, který je realizován s využitím knihovny JBox2D. Tato knihovna vychází z originální knihovny Box2D, která byla napsána v jazyce C. Fyzikální model pracuje s tělesy, které mají definované vlastnosti a interagují spolu v prostředí simulátoru. Každé těleso má hustotu, tření a míru odrazu. Dále má každé těleso definovaný tvar, který se může skládat z libovolného počtu obdélníků, kruhů nebo polygonů. Mezi tělesy dochází v prostředí simulátoru ke kolizím, které mají za následek změnu pozice nebo rychlosti. Tyto změny detailně sleduje tzv. řešitel, který během jednoho kroku, který proběhne v prostředí simulátoru, několikrát překontroluje a opraví pozice jednotlivých těles vůči dalším tělesům a jejich vektory rychlosti.

Programovací jazyk Java jsem zvolil pro vývoj samotné hry, protože jsem zvyklý s ním často pracovat a mám s ním větší zkušenosti, ve srovnání s ostatními jazyky.

1 Programovací jazyk Java

Jednou z hlavních výhod je přenositelnost mezi různými platformami díky virtuálnímu stroji Java Virtual Machine (dále jen JVM). Na rozdíl od programovacích jazyků C a C++ se nemusí programátor starat o dealokaci paměti, kterou za něj řeší tzv. garbage collector[1]. Zůstává tedy více prostoru pro samotný vývoj hry.

1.1 Rozhraní

V jazyce Java existuje rozhraní, které je definováno klíčovým slovem *Interface*. Toto rozhraní obsahuje definice všech metod, které musejí být implementovány třídou, která implementuje toto rozhraní. Rozhraní je tedy jakási univerzální šablona, kterou může implementovat jakákoli třída [3].

Rozhraní se může definovat například následovně:

```
public interface IAuto {
      void nastartuj();
      void odemkniDvere();
      boolean jeNastartovane();
}
```

Implementace rozhraní by potom mohla vypadat takto:

```
public class Auto implements IAuto {
      private boolean dvereOdemknuty = false;
      private boolean motorNastartovan = false;

      @override
      public void nastartuj() {
            motorNastartovan = true;
      }

      @override
      public void odemkniDvere() {
            dvereOdemknuty = true;
      }

      @override
      public boolean jeNastartovane() {
            return motorNastartovan;
      }
}
```

V programu se pak vytvoří objekt typu *IAuto*, do kterého se inicializuje objekt typu *Auto*, který implementuje rozhraní, jak je vidět na tomto příkladu:

```
/*
 * Hlavni program
 */
IAuto auto = new Auto();
```

[1] Garbage collector je obvykle část běhového prostředí jazyka, nebo přídavná knihovna, podporovaná kompilátorem, hardware, operačním systémem, nebo jakoukoli kombinací těchto tří. Má za úkol automaticky určit, která část paměti programu je už nepoužívaná, a připravit ji pro další znovupoužití [13].

```
auto.odemkniDvere();
if(!auto.jeNastartovane()) {
      auto.nastartuj();
}
```

Rozhraní se hojně využívá v kolekcích, které nabízí jazyk Java.

1.2 Kolekce

Java standardně nabízí implementaci různých typů datových struktur, se kterými je programování jakékoli aplikace o něco jednodušší, protože nemusíme programovat vlastní datové struktury [3].

Mezi jedny ze základních rozhraní kolekcí patří:

- *List* – seznam,
- *Queue* – fronta,
- *Set* – množina,
- a *Map* – tabulka.

Rozhraní *List* neboli seznam umožňuje vkládání prvku do datové struktury a umožňuje odebírání prvku na základě znalosti indexu.

Rozhraní *Queue* neboli fronta umožňuje vkládání prvku do datové struktury a umožňuje odebírání prvků na základě pořadí, v jakém byly do fronty vloženy. Uplatňuje se zde filozofie FIFO – First In, First Out.

Rozhraní *Set* neboli množina umožňuje vkládání prvku do datové struktury a umožňuje odebírání prvku na základě znalosti prvku samotného.

Rozhraní *Map* neboli tabulka umožňuje vkládání prvku jako páru, který se skládá z klíče a hodnoty, a umožňuje odebírání prvku na základě znalosti klíče, který je s ním v páru.

Jednotlivé rozhraní implementují konkrétní třídy realizující různé datové struktury. Mezi nejpoužívanější patří třída *ArrayList* implementující rozhraní *List* a třída *HashMap* implementující rozhraní *Map*.

Vytvoření seznamu a práce s ním může vypadat takto:

```
List<ClassType> seznam = new ArrayList<ClassType>(); // vytvoření seznamu
seznam.add(1); // vložení prvku s hodnotou „1“
seznam.add(3); // vložení prvku s hodnotou „3“
seznam.add(1); // vložení prvku s hodnotou „1“
System.out.println(seznam.size()); // vypíše „3“
```

Zde vidíme, že seznam uchovává všechny prvky, které do něj vložíme, a nestará se o duplicity, které tím vznikají.

Vytvoření tabulky a práce s ní může vypadat takto:

```
Map<Key, Value> tabulka = new HashMap<Key, Value>(); // vytvoření tabulky
```

```
tabulka.put(1, "ahoj"); // vložení prvku s klíčem „1"
tabulka.put(3, "nashledanou"); // vložení prvku s klíčem „3"
tabulka.put(1, "dobrý den"); // vložení prvku s klíčem „1"
tabulka.put(2, "ahoj"); // vložení prvku s klíčem „2"
System.out.println(tabulka.size()); // vypíše „3"
```

Zde vidíme, že tabulka nevkládá prvky, jejichž klíč *Key* je v tabulce již obsažen. Na druhou stranu tabulce nevadí duplicitní hodnoty *Value* vkládaného prvku.

Vytvoření množiny a práce s ní může vypadat takto:

```
Set mnozina = new TreeSet(); // vytvoření množiny
mnozina.add(1); // vložení prvku s hodnotou „1"
mnozina.add(3); // vložení prvku s hodnotou „3"
mnozina.add(1); // vložení prvku s hodnotou „1"
System.out.println(mnozina.size()); // vypíše „2"
```

Zde je vidět, že se množina chová tak, že neobsahuje duplicitní prvky, ale každý prvek je v ní právě jednou.

Vytvoření fronty a práce s ní může vypadat takto:

```
Queue fronta = new ArrayDeque(); // vytvoření fronty
fronta.add(1); // vložení prvku s hodnotou „1"
fronta.add(1); // vložení prvku s hodnotou „1"
fronta.add(3); // vložení prvku s hodnotou „3"
System.out.println(fronta.size()); // vypíše „3"
System.out.println(fronta.poll()); // vypíše „1"
```

Zde vidíme, že ve frontě mohou být duplicitní prvky. Také vidíme, že fronta opravdu zpřístupní prvek, který do ní byl vložen jako první.

1.3 Výčtový typ

Každý programátor se setkal s tím, že potřeboval konstantu, která by reprezentovala nějaké číslo, ale zároveň by bylo hned jasné, co které číslo znamená. Přesně pro tento účel Java standardně nabízí výčtový typ neboli *Enum* [3].

Definice výčtového typu je velice snadná. K vytvoření se používá klíčové slovo *enum.* Do výčtového typu vypíšeme slova, která budou automaticky reprezentována číslem, resp. ordinální hodnotou. Definice výčtového typu může vypadat takto:

```
public enum EStav {
      ZAPNUTO, VYPNUTO;
}
```

Pokud bychom chtěli, aby jednotlivé položky výčtového typu měli i návratovou hodnotu v podobě textu, definice by vypadala takto:

```
public enum EStav {
      ZAPNUTO {
            @Override
            public String toString() {
                  return "Přístroj je zapnuty.";
```

```
            }
        },
        VYPNUTO {
            @Override
            public String toString() {
                return "Přístroj je vypnuty.";
            }
        };
}
```

Výčtový typ lze velice prakticky využívat například v klauzuli *switch*, která je nedílnou součástí téměř každého programu. Jednoduchý příklad využití výčtového typu v klauzuli *switch* lze demonstrovat na třídě, která reprezentuje nějaký elektrický spotřebič, který signalizuje, jestli je zapnutý nebo vypnutý.

```
public class Spotrebic {
    private boolean EStav stav = EStav.VYPNUTO;

    public void Zapnout() {
        stav = EStav.ZAPNUTO;
    }

    public void Vypnout() {
        stav = EStav.VYPNUTO;
    }

    public boolean jeZapnuto() {
        switch(stav) {
            case ZAPNUTO:
                return true;
            case VYPNUTO:
                return false;
        }
    }
}
```

2 Herní vývojový nástroj

Slick2D je knihovna šířena pod licencí BSD. Původním autorem knihovny je Kevin Glass, ke kterému se později přidali další programátoři, kteří přidali do knihovny pár dalších funkcí [5].

Knihovna využívá ke své funkcionalitě částečně i knihovnu Lightweight Java Game Library (dále jako LWJGL) a poskytuje programátorovi rozhraní a rozmanité nástroje ke kompletnímu vývoji hry [4].

2.1 Knihovna Slick2D v prostředí NetBeans IDE

Knihovnu lze bezproblémově používat ve vývojovém prostředí NetBeans IDE. Nejdříve stáhneme potřebné komponenty z domovských stránek jednotlivých projektů:

- Slick2D (http://www.slick2d.org/downloads/slick.jar),
- LWJGL (http://www.lwjgl.org/download.php).

Po stažení obou knihoven, rozbalíme obsah do dočasné složky a do otevřeného projektu ve vývojovém prostředí NetBeans IDE přidáme knihovny:

- lwjgl.jar,
- Slick.jar,
- jinput.jar,
- a lwjgl_util.jar.

Ostatní soubory s příponou *.dll zkopírujeme do složky *natives*, kterou vytvoříme v kořenovém adresáři projektu.

Nakonec v nastavení projektu v položce *Build/Run* napíšeme do kolonky *VM Options* následující parametry: -Djava.library.path=./natives. Tím je nastaveno vyhledávání nativních knihoven a knihovna Slick2D je připravena k použití [6].

2.2 Herní cyklus

Herní smyčka je základ každé počítačové hry. Knihovna má herní smyčku skrytou před zraky programátora a poskytuje pouze rozhraní, které programátor implementuje [5].

Implementací základního herního rozhraní dostane programátor k dispozici tři základní metody, kolem kterých se točí celý životní cyklus hry:

- *init* – metoda, která je zavolána při startu programu a slouží k inicializaci veškerého obsahu, který bude v průběhu potřeba,
- *update* – metoda, která je opakovaně volána, dokud není hra ukončena, a slouží k aktualizaci herních mechanismů,
- a *render* – metoda, která je také volána opakovaně, dokud není hra ukončena, a slouží k vykreslení grafického obsahu hry [1][2].

Tyto tři metody jsou pro programátora her absolutní základ a knihovna Slick2D je také jako základ poskytuje. Tím to vlastně končí, protože kromě pomocných nástrojů, které může programátor využívat, musí programátor použít svoji fantazii a zkušenosti, k tomu aby vytvořil obstojnou hru.

2.3 Grafika

Žádná hra se neobejde bez grafických prvků, a proto v knihovně nechybí třídy pro práci s grafikou, resp. obrázky.

K dispozici jsou tři základní třídy:

- *Image* – třída pro práci s obrázkem (podpora formátů PNG, GIF, TGA a JPG),
- *SpriteSheet* – třída pro práci s obrázkem, který obsahuje více obrázků poskládaných v mřížce (např. rozfázovaná animace),
- a *Animation* – třída pro práci s animací [5].

Třída *Image* poskytuje všechny možné metody, které by se mohly hernímu programátorovi v průběhu vývoje hry hodit. Samozřejmostí jsou metody pro oříznutí obrázku, rotaci obrázku, zesvětlení a ztmavení obrázku, překrytí obrázku texturou nebo samotné vykreslení obrázku do grafiky. Použití třídy zobrazuje následující příklad.

```
Image img = new Image("obrazek.png");
```

Třída *SpriteSheet* se chová podobně jako třída *Image*, ale je u ní kladen důraz na to, že obrázek v sobě uchovává více obrázků, které jsou rovnoměrně rozprostřené po celé ploše obrázku. Výhoda oproti používání třídy *Image* je ta, že program inicializuje pouze jeden obrázek místo dvaceti obrázků (kvůli kterým by program musel inicializovat dvacet objektů). Použití třídy, kde velikost pod-obrázku je 40 pixelů na šířku a 60 pixelů na výšku:

```
SpriteSheet sheet = new SpriteSheet("spritesheet.png", 40, 60);
```

Třída *Animation* je jakási nadstavba, která umí pracovat se třídou *Image* i se třídou *SpriteSheet* a zajišťuje změnu obrázku, po předem definovaných časových intervalech. Třída šetří paměť tím, že pracuje s obrázky, které jsou již inicializované. Animace se automaticky aktualizuje při každém vykreslení dalšího snímku v metodě *render*. Tato funkce jde vypnout a programátor může nechat aktualizovat animaci ručně v metodě *update*. Použití třídy, kde se animace skládá ze tří obrázků a aktualizace animace proběhne po 250 ms:

```
Animation anim = new Animation(new Image[]{img1, img2, img3}, 250);
```

2.4 Vstupy a výstupy

Knihovna Slick2D umí spravovat standardní vstupy z klávesnice a myši, ale také vstupy z různých herních ovladačů. K tomu využívá třídu *Input*, která je nabídnuta programátorovi skrz herní kontejner, tak že má programátor stále přístup ke všem vstupům [5].

První možnost přístupu ke vstupům je skrz herní kontejner, který poskytuje pomocí metody instanci třídy *Input*.

Druhá možnost je, že objekt implementuje rozhraní *InputListener*, *KeyListener* nebo *MouseListener* a zaregistruje se, aby byl aktivní. Pomocí herního kontejneru se zpřístupní instance třídy *Input*, která nabízí možnost zaregistrovat objekt, který implementuje jedno z výše zmíněných rozhraní [5].

Třída *Input* nabízí programátorovi širokou nabídku využití. Mezi nejpoužívanější funkce patří:

- detekce stisknutí klávesy – klávesa jde dolů,
- detekce uvolnění klávesy – klávesa jde nahoru,
- detekce stisknuté klávesy – klávesa je dole,
- detekce stisknutí tlačítka na myši – tlačítko jde dolů,
- detekce uvolnění tlačítka na myši – tlačítko jde nahoru,
- detekce stisknutého tlačítka na myši – tlačítko je dole,
- detekce změny polohy kurzoru,
- detekce stisknutého tlačítka na myši a změny polohy kurzoru,
- detekce změny polohy kolečka na myši,
- a detekce kliknutí tlačítka na myši – tlačítko jde dolů a zase nahoru.

Díky tomu lze snadno obsluhovat i několik kláves současně a využívat různé události myši. Metody obsluhující události myši většinou poskytují polohu kurzoru a číslo tlačítka, které je právě stisknuto nebo uvolněno, nebo poskytují informaci o tom, zdali byl proveden dvojklik.

2.5 Zvuky a hudba

Správa zvuků a hudby je velice podobná jako správa obrázků pomocí třídy *Image*. Pro práci se zvuky je v knihovně třída *Sound*, a pro práci s hudbou je v knihovně třída *Music*. Obě dvě třídy využívají ke své práci knihovnu OpenAL, kterou poskytuje knihovna LWJGL [5].

Zvuky mohou být spouštěny nezávisle na sobě, a tím pádem i přes sebe. Používají se k tomu základní metody pro spuštění, vypnutí a opakování, aby se zvuk opakoval stále dokola, dokud nebude opět vypnut. Oproti tomu hudba může být spuštěna vždy jen jedna. Stejně jako zvuky je i hudba ovládána pomocí základních metod pro spuštění, vypnutí nebo opakování. Samozřejmostí je možnost nastavení hlasitosti, ale i nastavení základní výšky zvuku nebo hudby. Zvuky běží stejně jako hudba na samostatném kanálu, tak aby se navzájem neovlivňovaly. Tím je dosaženo lepší kvality ozvučení [5].

Použití třídy *Sound*:

```
Sound effect = new Sound("click.ogg");
effect.play();
```

Použití třídy *Music*:

```
Music music = new Music("track1.ogg");
music.loop();
```

Třídy podporují zvukové formáty WAV a OGG. Soubor typu WAV nese informaci o nekomprimované zvukové vlně v plné kvalitě a tomu také odpovídá jeho velikost. Pro představu: 5 minut dlouhý zvukový záznam uložený ve formátu WAV, má přibližně 50 MB. Soubor typu OGG nese informaci o zvukové vlně komprimovanou, a proto je výsledná velikost souboru několikanásobně menší. Pro představu: 5 minut dlouhý zvukový záznam uložený ve formátu OGG, má přibližně 5 MB.

Rozdíl ve velikosti souborů ve formátu WAV a OGG je výrazný a proto je, z hlediska úspory místa, vhodnější volit formát OGG. Velikost většiny her udává právě velikost grafického a zvukového obsahu hry, než samotný kód hry. Z tohoto důvodu je výhodnější zvolit zvukové stopy ve formátu OGG.

2.6 Podpora fontů

Knihovna podporuje práci s fonty a definuje rozhraní *Font*, které implementují třídy podle typu fontu:

- *UnicodeFont* – umí pracovat s fontem ve formátu TrueTypeFont,
- *SpriteSheetFont* – využívá k vytvoření fontu třídu pro obrázky *SpriteSheet*,
- *AngelCodeFont* – pracuje s obrázkem, ve kterém jsou znaky a datovým souborem, který popisuje polohu jednotlivých znaků. Nevýhoda je, že použití může vyvolat lehké zpomalení celé hry [5].

Použití třídy *UnicodeFont* je velice snadné, jak je vidět na příkladu:

```
UnicodeFont font = new UnicodeFont("font.ttf", 12, false, false);
font.addAsciiGlyphs();
font.getEffects().add(new ColorEffect()); // obarví písmo na bílo
font.loadGlyphs();
g.setFont(font);
```

Použití třídy *AngelCodeFont* je také jednoduché, ale pro každý font se musí vytvořit dva soubory – obrázek se znaky a soubor popisující polohu jednotlivých znaků. K tomu se využívá aplikace, která tyto dva soubory umí generovat na základě nějakého fontu typu TrueTypeFont. Pokud chce programátor ve své hře využít jeden font, ale v různé velikosti – např. 12 a 18 – musí pro každou velikost vytvořit pomocí generátoru dva soubory. Z mého pohledu je to trochu těžkopádné a mnohem výhodnější shledávám použití třídy *UnicodeFont*, která dokáže pracovat s fontem typu TrueTypeFont a měnit velikost fontu při inicializaci – tím odpadá externí generování dvou souborů pro každou velikost fontu [5].

Nastavení barvy fontu je stejné jako při práci se standardními knihovnami:

```
g.setColor(Color.white);
g.drawString("nejaky text", x, y);
```

2.7 Stavově koncipovaná hra

Základem každé modernější hry je to, že je hra rozdělená na různé situace, resp. stavy. Je to vlastně logické, protože k čemu inicializovat, aktualizovat a vykreslovat hru, resp. herní režim, když se hráč nachází teprve v hlavním menu.

Hra se rozdělí na různé stavy – např. úvod, hlavní menu, herní režim, herní menu – a program, dle algoritmů napsaných programátorem, přepíná jednotlivé stavy. Tím je dosaženo to, že program aktualizuje a vykresluje pouze aktivní situaci, resp. stav a ostatní stavy jsou uspané – neprobíhá u nich aktualizace ani vykreslování – a tím se šetří výpočetní výkon.

2.8 Knihovna Lightweight Java Game Library

Knihovna LWJGL (LightWeight Java Game Library) je řešení pro profesionální i amatérské programátory Javy, kteří chtějí programovat svou vlastní hru. Knihovna poskytuje vývojářům přístup k vysoce výkonným knihovnám jako OpenGL (Open Graphics Library), OpenCL (Open Computing Language) a OpenAL (Open Audio Library), které umožňují nejmodernější 3D hry a 3D zvuk [7].

2.9 Knihovny Jorbis a JOGG

Knihovny Jorbis a JOGG se používají pro dekódování zvukových stop, které jsou ve formátu OGG. Ogg je sám o sobě projekt, který si klade za cíl vytvořit svobodný software pro digitální multimédia [8].

OGG jako datový formát se chová pouze jako kontejner, který dokáže uchovat audio formát, video formát, textový kodek[2] nebo titulkové struktury. Audio formát je často tvořen ztrátovou kompresí Vorbis nebo bezeztrátovou kompresí FLAC.

Zvukové stopy zakódované v kontejneru Ogg jsou hojně využívány při vývoji her právě proto, že se jedná o svobodný software šířený pod licencí BSD [8].

---

[2] Kodek (složenina z počátečních slabik slov **ko**dér a **dek**odér) je zařízení nebo počítačový program, který dokáže transformovat datový proud nebo signál [14].

3 Fyzikální model hry

Knihovna JBox2D vznikla přenesením kódu do Javy z originálu Box2D, který byl napsán v jazyce C++, jehož autorem je Erin Catto. JBox2D je šířen pod stejnou licencí jako Box2D a to pod licencí zlib. Zakladatelem projektu JBox2D je Daniel Murphy, který začal přepisovat originální kód z C++ do Javy v roce 2007. Aktuální poslední verze JBox2D 2.1.2.3 je ze dne 18. února 2013 – jedná se pouze o opravu některých chyb [9].

JBox2D je simulátor fyziky a pracuje s tzv. systémem KMS – jako jednotku váhy používá kila, jako jednotku délky používá metry a jako délku času používá sekundy. Simulátor simuluje působení sil mezi jednotlivými pevnými tělesy – je to vlastně pouze matematický model, který pracuje s předem definovanými objekty a poskytuje výstup v podobě čísel, se kterými programátor může udělat, co potřebuje [9].

Autor originálu Box2D upozorňuje v manuálu, že ten kdo bude s knihovnou – simulátorem – pracovat, by měl být obeznámen se základními principy fyziky – tj. hmotnost, síla, točivý moment a impuls. Též doporučuje dobrou znalost programovacího jazyka C++, v našem případě tedy dobrou znalost programovacího jazyka Java [10].

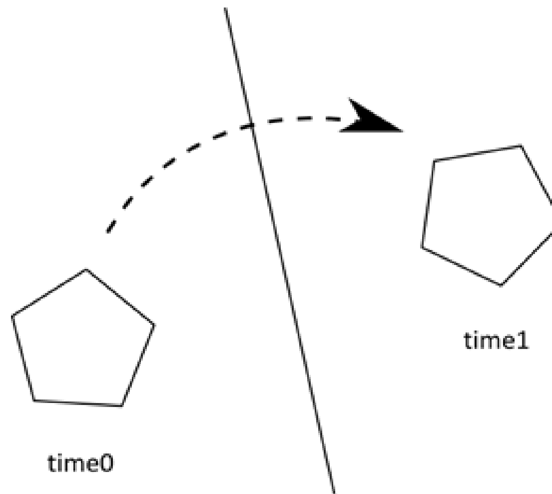
Manuál ke knihovně JBox2D neexistuje, ale jeho autor Daniel Murphy doporučuje manuál napsaný k originální knihovně Box2D, protože veškeré principy jsou totožné. Tedy i já jsem z originálního manuálu vycházel, když jsem se učil s knihovnou JBox2D pracovat. V manuálu nejsou bohužel odkryty úplně všechny vlastnosti a nástroje, ale pouze základní principy popisující práci se simulátorem – tedy jak vytvořit prostředí, objekty, spojení mezi nimi, apod.

3.1 Základní koncept

Knihovna pracuje s několika základními objekty:

- tvar (anglicky shape) – dvourozměrný geometrický objekt (např. obdélník nebo kruh),
- pevné těleso (anglicky rigid body) – kus hmoty, která je tak pevná, že vzdálenost mezi dvěma kousky hmoty je konstantní (v originálním manuálu se píše doslova, že je hmota tvrdá jako diamant),
- vlastnosti (anglicky fixture) – definice vlastností, které vážou určitý tvar k pevnému tělesu a definují další klíčové vlastnosti jako hustotu, tření a míru odrazu,
- omezení (anglicky constraint) – fyzické spojení, které odebírá pevným tělesům určitou volnost (2D těleso má tři typy volnosti – pohyb po ose X, pohyb po ose Y a možnost rotace),
- kontaktní omezení (anglicky contact constraint) – zajišťuje, že pevná tělesa nepronikají sami sebou a stanovuje mezi jednotlivými tělesy tření a míru odrazu; kontaktní omezení je automaticky vytvářeno simulátorem, tedy není potřeba ho vytvářet ručně,

- spojení (anglicky joint) – definuje mezi pevnými tělesy jistý druh spojení, tzn. že jsou například fyzicky spojeny a drženy v předem definované vzdálenosti,
- limit spojení (anglicky joint limit) – definuje jisté omezení v konkrétním typu spojení mezi pevnými tělesy (např. omezuje úhel otočení, apod.),
- motor spojení (anglicky joint motor) – řídí pohyb konkrétního spojení mezi pevnými tělesy podle volnosti, jakou mají (např. rotace tělesa určitou rychlostí),
- svět (anglicky world) – prostředí simulátoru, ve kterém spolu interagují všechny objekty (pevná tělesa a jejich vlastnosti, omezení a spojení),
- řešitel (anglicky solver) – prostředí simulátoru používá řešitel, který má na starosti postup času a řešení kontaktů objektů a všech omezení,
- kontinuální kolize (anglicky continuous collision) – řešitel počítá postup pevných těles v čase pouze v diskrétních časových krocích – může tak tedy dojít k nežádoucímu efektu, který se nazývá tunelování – jak je vidět na následujícím obrázku (Obrázek 1 – Časové tunelování) [11].



Obrázek 1 – Časové tunelování

Simulátor obsahuje speciální algoritmy pro řešení tunelování. První algoritmus umí interpolovat pohyb dvou těles a díky tomu najít časový okamžik první srážky (anglicky TOI – Time Of Impact). Druhý algoritmus vytvoří mezikroky, díky kterým detekuje okamžik první srážky mezi dvěma pevnými tělesy a vyřeší kolizi [11].

3.2 Moduly

Knihovna, resp. simulátor se skládá ze tří modulů: *Common*, *Collision*, a *Dynamics*.

- *Common* – obsahuje funkci pro alokaci, matematické funkce a veškeré nastavení,
- *Collision* – definuje tvary a obsahuje funkce obsluhující kolize,
- *Dynamics* – poskytuje simulaci fyzického světa a pevných těles v něm, včetně veškerých dalších základních objektů jako spojení, omezení, atd. [11].

3.3 Jednotky

Již bylo zmíněno, že simulátor pracuje se systémem KMS – tedy jednotkou váhy je kilo, jednotkou délky je metr a jednotkou času je sekunda.

Autor doporučuje volit velikost objektů v rozmezí mezi 0,1 a 10 metrů, protože to zaručuje spolehlivý chod simulace. Objekty menší nebo větší mohou způsobit nepředvídatelné chování simulátoru. Doporučená velikost objektů tedy odpovídá nejmenšímu objektu – např. konzervě – a největšímu objektu – např. autobusu. Tyto objekty by spolu měli interagovat naprosto v pořádku [11].

K vykreslování objektů, se kterými pracuje simulátor, je potřeba stanovit nějaký převodní systém mezi jednotkami simulátoru – metry – a jednotkami zobrazovacího zařízení – pixely. V žádném případě by neměl simulátor pracovat přímo s pixely, protože největší herní objekt, který by se mohl v simulátoru objevit, by měl pouhých 10 pixelů, a naopak nejmenší možný objekt by měl pouze 1 pixel.

Převodní systém můžeme tedy stanovit například tak, že 64 pixelů bude 1 metr. Tím pádem 1 pixel bude 0,015625 metru – a simulátor bude pracovat ve správném rozmezí [11].

3.4 Továrny a definice

Správa paměti hraje hlavní roli v návrhu Box2D API – v JBox2D by to tedy mělo být obdobné. Když chceme vytvořit pevné těleso, které je reprezentované třídou *Body* nebo spojení, které je reprezentované třídou *Joint*, musíme zavolat tovární funkci, kterou nám poskytuje třída *World*. Žádným jiným způsobem by se tyto objekty neměly vytvářet [11].

Metody pro vytváření pevných těles a spojení vypadají následovně:

```
// metody třídy World
public Body CreateBody(BodyDef bd);
public Joint CreateJoing(JointDef jd);
```

Podobně vypadají i metody pro vytváření vlastností těles:

```
// metody třídy Body
public Fixture CreateFixture(FixtureDef fd);
```

3.5 Uživatelská data

Třídy *Body*, *Joint* a *Fixture* umožňují nést jakýkoli uživatelem definovaný objekt. Herní programátor může tedy objektům těchto tříd přiřadit určitá herní data, která reprezentují herní objekt [11].

Při vytváření instance některé z těchto tří tříd definujeme, s jakým typem uživatelských dat bude instance pracovat.

```
// ...
// vytvoření pevného tělesa pomoci továrny
```

```
Body<UserData> body = World.CreateBody(BodyDef bd);
// ...
body.setUserData(userData);
```

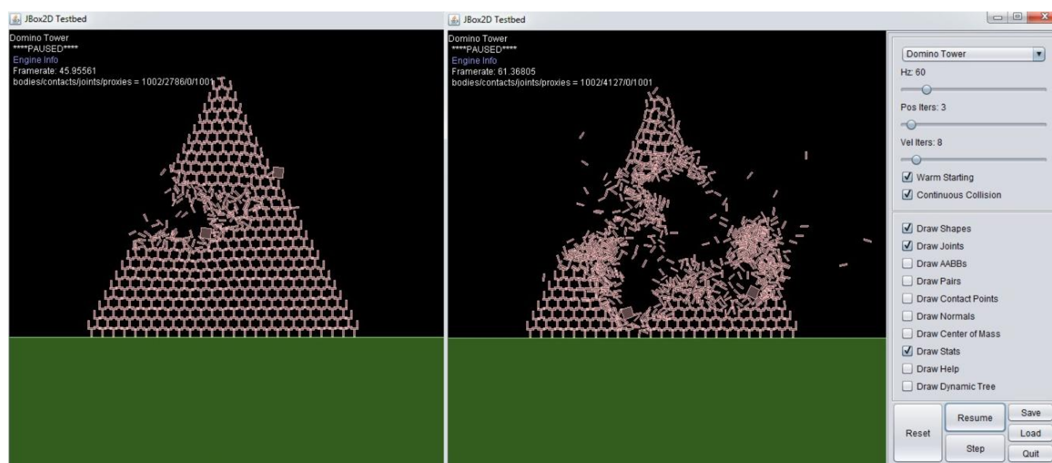
3.6 Simulátor fyzikálního modelu

Simulátor fyzikálního modelu – tzv. svět, obsahuje všechny objekty simulace – pevná tělesa, vlastnosti tělesa, spojení a omezení. Ke správnému fungování celé simulace využívá simulátor tzv. řešitele omezení. Ten v každém jednotlivém časovém úseku řeší omezení, avšak vždy jen jedno v daný moment – jakmile omezení vyřeší, již se k němu nevrací a pokračuje k řešení dalšího omezení. Všechny omezení spolu nějak souvisejí, protože objekty v simulaci spolu interagují, proto pouze jedno jediné omezení je vyřešeno dokonale. Je to způsobeno tím, že vyřešení jednoho omezení může narušit řešení jiného omezení. Chceme-li v každém časovém kroku simulace získat dobré řešení všech omezení, musíme provést více iterací řešitele omezení, tzn. že se každé řešení v rámci jednoho kroku řeší vícekrát [11].

Řešitel omezení pracuje ve dvou fázích: v první řeší rychlost objektů a v druhé jejich pozici. V první fázi počítá řešitel impulsy, které jsou nezbytné pro správný pohyb objektů, zatím co v druhé fázi se snaží o snížení překrývání objektů tím, že mění jejich pozici [11].

Počet iterací první i druhé fáze se může lišit. Doporučený počet, uváděný autorem, je 8 iterací pro rychlost a 3 iterace pro pozici. Počet iterací se samozřejmě odráží na výkonu. Vyšší počet iterací znamená lepší přesnost, ale může mít za následek zpomalení hry. Je tedy potřeba najít ten správný kompromis [11].

Interakce objektů ve fyzikálním modelu je vidět na obrázku (Obrázek 2 – Demonstrace knihovny JBox2D), který znázorňuje dva časové úseky.



Obrázek 2 – Demonstrace knihovny JBox2D

3.7 Pevné těleso a jeho vlastnosti

Pevné těleso v simulaci dělíme na tři základní typy:

- statické – zůstává po celou dobu simulace na jednom místě, chová se jako by mělo nekonečnou hmotnost a nemůže kolidovat s ostatními statickými nebo kinematickými tělesy,
- kinematické – chová se podobně jako statické těleso, ale na rozdíl od něj může mít určitou rychlost, resp. směr, kterým putuje, ale nepůsobí na něj žádné jiné síly,
- dynamické – plně simulované těleso, na které působí všechny síly v simulátoru a chová se podle předem definovaných vlastností.

U všech tří typů může programátor ručně nastavit jejich pozici, která se hned v následujícím časovém kroku projeví [11].

Vlastnosti pevného tělesa definují jeho chování vůči ostatním tělesům v simulaci. Vlastnosti lze rozdělit:

- hustota – používá se pro výpočet hmotnosti pevného tělesa,
- tření – určuje, jak snadno se pevné těleso pohybuje vůči druhému tělesu v těsné blízkosti,
- a míru odrazu – určuje elasticitu srážky mezi dvěma tělesy [11].

Hustota těles by se neměla numericky příliš lišit – příliš velké číselné rozpětí by mohlo způsobit nestabilitu simulace. Nicméně hustota může nabývat hodnot od nuly do jakékoli kladné hodnoty [11].

Tření by mělo nabývat hodnot mezi 0 a 1, ale může nabývat jakékoli kladné hodnoty. Hodnota 0 úplně vypne tření, takže se dvě tělesa v těsném kontaktu pohybují bez vzájemného ovlivnění rychlosti pohybu. Naopak hodnota 1 způsobí výrazné zpomalení pohybu dvou těles [11].

Míra odrazu by měla nabývat také hodnot mezi 0 a 1. Pokud by těleso, u kterého se definuje míra odrazu, byl míč, tak by hodnota 0 znamenala, že při dopadu na zem se neodrazí a naopak hodnota 1 by znamenala, že se bude odrážet do nekonečna [11].

Pevné těleso je reprezentováno třídou *Body*, která se definuje pomocí objektu typu *BodyDef*. Všechny vlastnosti se nastavují pomocí objektu typu *FixtureDef*, ze kterého se vytvoří objekt typu *Fixture* a ten se přiřadí objektu typu *Body* [11].

3.8 Typy spojení

Spojení definuje mezi dvěma a více pevnými tělesy určité omezení, které má vliv na jejich pohyb. Již bylo zmíněno, že každé pevné těleso má v simulaci tři typy volnosti – pohyb po ose X, pohyb po ose Y a možnost rotace. Spojení může některý typ volnosti zcela nebo částečně omezit [11].

Do přeneseného kódu JBox2D se bohužel nedostaly úplně všechny typy spojení z originálního Box2D, ale i tak v knihovně existuje přes deset typů spojení. Z tohoto

důvodu budou v této části popsány pouze ty důležitější a hojně využívané typy. Patří mezi ně:

- vzdálené spojení (anglicky distance joint) – definuje konstantní vzdálenost mezi dvěma pevnými tělesy,
- a otočný kloub (anglicky revolute joint) – definuje bod, který ukotvuje pevné tělesa k jinému pevnému tělesu; vzájemně se mohou otáčet.

U vzdáleného spojení lze nastavit určitou míru elasticity. U otočného kloubu lze omezit úhel otočení. V originální knihovně Box2D existuje mj. provazové spojení (anglicky rope joint), které se chová jako opravdový provaz – má svou maximální délku, ale může se libovolně zmenšovat. Oproti tomu má vzdálené spojení délku pevně danou a může maximálně do jisté míry pružit. Bohužel toto provazové spojení se do Javové verze JBox2D nedostalo [11].

Spojení se vytváří pomocí objektu typu *Joint*, který se definuje pomocí objektu typu *JointDef*. Veškeré vytváření spojení, vlastností nebo pevných těles mají na starosti továrny, jak již bylo zmíněno v kapitole 3.4 [11].

3.9 Knihovna Fizzy

Na závěr části věnující se knihovně JBox2D, bych ještě rád představil knihovnu Fizzy, která částečně obaluje knihovnu JBox2D a jejímž autorem je Kevin Glass (autor knihovny Slick2D), který komunitě poskytuje tuto knihovnu pod licencí BSD. Jeho záměrem bylo poskytnout jednodušší a snadněji použitelné rozhraní pro práci s knihovnou JBox2D. Tím všem poskytl nástroj, který před nimi skrývá zdlouhavé opakující se definice a nabízí rychlé vytváření všech základních objektů. Neobsahuje však například rozhraní pro vytváření spojení [12].

4 Implementace logické 2D hry

V této kapitole jsou popsány klíčové části 2D logické hry, kterou jsem pracovně nazval Isaac. Název by měl hráči napovědět, že je ve hře využit nějaký fyzikální model, který dotváří celou logiku hry.

Hlavní postavou hry je robot Isaac, který se ocitnul v neznámém prostředí na neznámé planetě a nic si nepamatuje. Prochází herní úrovní a je nucen využít všech dostupných objektů k překonání nejrůznějších překážek. K manipulaci s objekty využívá své robotické rameno. Celá cesta herní úrovní je ztížena několika faktory: úroveň energie baterií, maximální počet předmětů, se kterými je možné manipulovat, maximální výkon. Na konci herní úrovně se setká s prvním obyvatelem planety, který mu zadá úkol, jehož splnění je podmínkou pro úspěšné dokončení herní ukázky.

4.1 Adresářová struktura hry

V kořenovém adresáři aplikace je soubor *run.bat*, který slouží ke správnému spuštění aplikace a soubor *Isaac_DEMO.jar*, který v sobě nese hlavní herní data a kód vlastní aplikace. Ve složkách *lib* a *natives* jsou externí knihovny.

```
\Kořenový adresář\
│  Isaac_DEMO.jar
│  readme.txt
│  run.bat
│
├───lib
│     JBox2D_2.1.2.3.jar
│     jinput.jar
│     jogg-0.0.7.jar
│     jorbis-0.0.15.jar
│     lwjgl.jar
│     lwjgl_util.jar
│     slick.jar
│
└───natives
      jinput-dx8.dll
      jinput-dx8_64.dll
      jinput-raw.dll
      jinput-raw_64.dll
      lwjgl.dll
      lwjgl64.dll
      OpenAL32.dll
      OpenAL64.dll
```

4.2 Rozdělení hry na stavy

V kapitole 2.7 bylo vysvětleno, co je to situačně koncipovaná hra. Nyní se podíváme, na jaké základní stavy je rozdělena hra Isaac:

- inicializace,
- hlavní menu,
- a herní režim.

Mezi jednotlivými stavy lze libovolně přepínat za běhu samotné aplikace. Také lze herní stav inicializovat, aniž by musel být aktivován. Herní stav je třída, která dědí ze třídy *BasicGameState*. Tyto jednotlivé stavy jsou uchovávány v hlavní herní třídě *Game*, která je spravuje. Třída dědí ze třídy *StateBasedGame*. Tato instance je klíčová při vytváření instance objektu *AppGameContainer*, který jí požaduje ve svém konstruktoru, jak je vidět v následujícím kusu kódu:

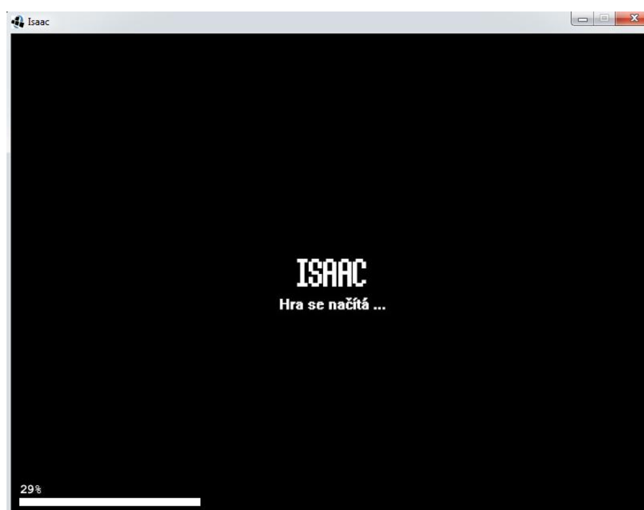
```
public static void main(String[] args) throws SlickException {
      AppGameContainer app = new AppGameContainer(new Game());
      app.setDisplayMode(800, 600, false);
      app.setTargetFrameRate(40);
      app.setTitle("Isaac");
      app.start();
}
```

Rodičovské třídy jsou poskytnuty knihovnou Slick2D. Ve třídě *Game* je klíčový konstruktor, který vytvoří a aktivuje první stav – inicializaci:

```
public static final int LOADING_STATE = 0;

public Game() {
      super("Isaac");
      this.addState(new LoadingState(LOADING_STATE));
      this.enterState(LOADING_STATE);
}
```

V prvním okamžiku, když se hra spustí, je tedy hra ve stavu inicializace. Je to prvotní stav hry, ve kterém dojde k inicializaci všech grafických a zvukových součástí – soubory formátu JPEG, PNG, OGG, atd. Moderní datově rozsáhlé hry provádí tuto inicializaci až těsně před spuštěním herního režimu, aby samotné spuštění hry – jako aplikace – netrvalo příliš dlouho. Nicméně hra Isaac je, proti dnešním herním gigantům, svou velikostí skromná, a proto probíhá inicializace potřebných dat hned ze začátku při spuštění aplikace.

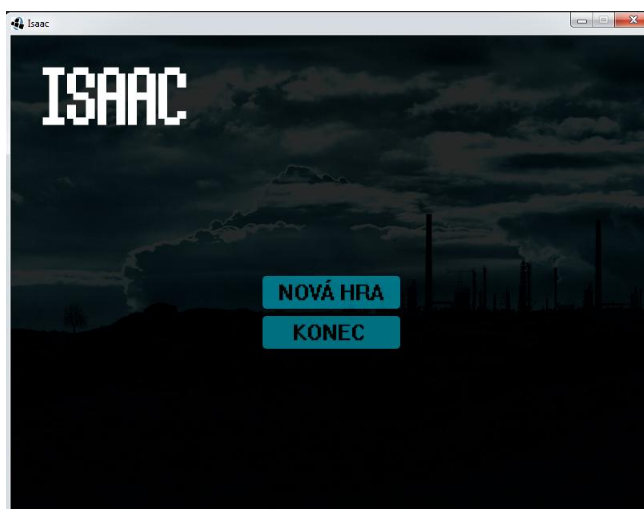


Obrázek 3 – Průběh načítání hry

Při inicializaci herních dat jsou vytvářeny objekty typu *Image*, *Sound* a *Music* (popsáno v kapitolách 2.3 a 2.5). Tyto objekty nesou herní data, která jsou připravena k okamžitému použití – např. vykreslení obrázku nebo přehrávání zvukové stopy. Kdyby herní data nebyly od začátku načteny, mohly by při hře vznikat nepříjemné časové prodlevy (snížení počtu snímků za sekundu), které by byly způsobené stálým načítáním herních dat v průběhu hry. To je jistě nežádoucí efekt, který kazí celkový dojem hry.

Poměr načtených herních dat k těm nenačteným, indikuje při spuštění hry bílý pruh, v dolní části obrazovky, spolu s číslem, které udává onen poměr v procentech. Samotná inicializace herních dat je detailněji popsána v následující kapitole 4.2.1.

Hlavní menu se aktivuje automaticky okamžitě po dokončení inicializace. Tedy hra se přepne ze stavu inicializace do stavu hlavního menu. Z hlavního menu vede jediná cesta ke spuštění hry – aktivace herního režimu – a to přes tlačítko s popisem *nová hra*. Herní menu je zobrazeno na následujícím obrázku (Obrázek 4 – Úvodní obrazovka s herním menu).



Obrázek 4 – Úvodní obrazovka s herním menu

Herní režim využívá načtených herních dat, které jsou uchovávány v kolekcích – převážně v objektu typu *Map*. Herní režim je nejnáročnější část programu, protože při aktualizaci hry probíhá nespočetně mnoho výpočtů, které potřebuje fyzikální model. Po každé aktualizaci přichází vykreslení všech herních objektů.

4.2.1 Inicializace herních dat

Herní data – zvuky a obrázky – jsou při vývoji ve stejném adresáři jako zdrojové soubory (obvykle ve složce *src*) a vývojové prostředí k nim tak přistupuje. Ve chvíli kdy se zkompilují zdrojové soubory a vytvoří se z nich archiv JAR a umístí se mimo adresář projektu, musí se k herním datům přistupovat jinou cestou, protože by je aplikace nenašla.

Při inicializaci herních dat proběhne v programu dotaz, zdali je aplikace spuštěna ve vývojovém prostředí či nikoli a podle toho zvolí správnou metodu pro načtení dat. Pokud se jedná o vývojové prostředí, tak algoritmus prochází postupně adresáře, ve kterých následně hledá soubory a pokud nalezne další adresář, přidá ho mezi prohledávané

adresáře a to se stále opakuje, dokud není prohledán poslední adresář. Pokud je aplikace spuštěna mimo vývojové prostředí, algoritmus pouze iteruje přes všechny soubory v archivu JAR.

Načtení dat probíhá tak, že algoritmus prohledává postupně do hloubky složky *images*, *sounds* a *music* a cesty k nalezeným souborům ukládá do instance objektu typu *ArrayList*. Po ukončení prohledávání se uloží do privátní proměnné počet nalezených souborů. Následuje postupná inicializace objektů typu *Image*, *Sound* a *Music* – podle toho z jaké složky soubor je – a při tom se uživateli na obrazovku graficky zobrazuje kolik procent souborů je již inicializováno.

4.3 Řízení hry

Třída *GameplayState* reprezentuje herní režim a obsahuje privátní statický atribut typu *GameController*, který má na starosti celé řízení hry, a poskytuje ho pomocí veřejné statické metody, aby k němu mohly přistupovat všechny ostatní objekty. Tento herní ovladač má klíčové atributy reprezentující:

- hráče – *Player* (popsáno v kapitole 4.6),
- kontejner herních úrovní – *Map<Integer, Ilevel>*,
- identifikační číslo aktivní herní úrovně,
- kontejner systémových zpráv – *SystemMsgContainer*,
- kontejner dialogů – *DialogContainer*,
- kontejner úkolů – *QuestContainer*,
- a kontejner použitelných fontů – *FontContainer*.

Místo atributu typu *ILevel*, který by reprezentoval herní úroveň, je použit kontejner, který může uchovávat více úrovní najednou a hráč tak mezi nimi může přepínat.

Jednotlivé třídy a rozhraní jsou popsány v následujících kapitolách.

4.4 Třídy herních prvků

Hra obsahuje mnoho tříd, které se starají o dílčí práci. Mezi takové prvky patří různé kontejnery, továrny nebo třídy zajišťující určitou specifickou funkci.

Další důležitou částí jsou výčtové typy. Mezi jeden z nejdůležitějších výčtových typů patří *EObjectType*, který definuje typ herního objektu – např. hráč, věc, neviditelný statický objekt, NPC, atd. Dalším důležitým výčtovým typem je *ESimpleObject*, který definuje všechny jednoduché stvořitelné herní objekty (celkem 30) – např. železná bedna, dřevěná bedna, atd. Výčtový typ *EFont* definuje typy fontů, které je možné ve hře použít. *EItemObject* definuje všechny předměty, které může hráč dát do inventáře. Výčtový typ *ELevel* definuje názvy všech plánovaných herních úrovní (celkem 6), ale v hratelné ukázce je zpřístupněna pouze první.

4.4.1 Kontejnery

Herní ovladač (popsáno v kapitole 4.3) pracuje s kontejnery, mezi které patří také *DialogContainer* a *SystemMsgContainer*. Tyto dvě třídy mají společného rodiče, kterým je třída *TextContainer*, která obsahuje většinu kódu. Obsahuje frontu zpráv a čítače, které zajišťují odebírání zpráv a nastavování aktuálně zobrazené zprávy. Časovače se aktualizují pomocí metody *update()*, která je volána s každou aktualizací herní smyčky. Pro úsporu výpočetního výkonu může textový kontejner přejít do režimu spánku, když neobsahuje žádnou zprávu a být opět probuzen při příchodu nové zprávy. *DialogContainer*, jak již název napovídá, je určen pro zobrazování dialogu – titulků – které jsou pro hráče klíčové. *SystemMsgContainer* se stará o zobrazování systémových zpráv, které jsou pouze informativního charakteru – např. že hráč sebral předmět, nebo že dosáhl záchytného bodu.

Třída *FontContainer* je kontejner instancí použitelných fontů. Je to z důvodu, aby se při každé potřebě použití jiného fontu nemusela inicializovat nová instance objektu, který font poskytuje. Kontejner při svém vzniku inicializuje všechny potřebné fonty a pomocí metod na ně poskytuje reference. Fonty jsou potřeba na několika místech v kódu při každé aktualizaci hry (aktualizace proběhne čtyřicetkrát za sekundu) a opakovaná inicializace by se mohla projevit na rychlosti hry. Ve hře je použit font Free Sans, který je volně šiřitelný.

Jeden z posledních významných kontejnerů je třída *QuestContainer*. Při vytváření scénáře hry jsou do tohoto kontejneru vloženy všechny herní úkoly, které může hráč splnit. Úkoly jsou uchovány v kolekci *Map*, kde klíčem je NPC postava a hodnotou je samotný úkol. V následujícím kusu kódu je vidět metoda pro vkládání úkolu a zpřístupňování úkolu:

```
private Map<NPC, Quest> quests;

public void createQuest(Quest quest, NPC npc) {
      quests.put(npc, quest);
}

public Quest getQuest(NPC npc) {
      return quests.get(npc);
}
```

Ve chvíli kdy se hráč přiblíží k NPC postavě, mu je zadán úkol, který je spojen s touto postavou.

4.4.2 Továrny

Program využívá několik továren, kvůli zjednodušení vytváření objektů, které jsou často používány.

Třída *JointFactory* se stará o vytváření spojení (popsáno v kapitole 3.8) mezi pevnými tělesy ve fyzikálním simulátoru. Obsahuje metody zajišťující samotnou definici spojení a následovné vytvoření, jak je vidět například v následujícím kusu kódu:

```
public static DistanceJoint createDistanceJoint(
            Body a, Body b, Vec2 anchorA, Vec2 anchorB) {
      DistanceJointDef djd = new DistanceJointDef();
      djd.initialize(a, b, anchorA, anchorB);
```

```
        djd.collideConnected = true;
        Joint j = world.createJoint(djd);
        GameplayState.getGameController().getActiveLevel().addJoints(j);
              return (DistanceJoint) j;
}
```

Třída *SimpleObjectFactory* se stará o vytváření jednoduchých herních objektů. Funkci zajišťuje jediná statická metoda, která pomocí přepínače přepíná výčtový typ, který určuje, který objekt má být vytvořen.

```
public static ILevelObject getSimpleObject(
            float x, float y, ESimpleObject sgo) throws SlickException {
      switch (sgo) {
            case METAL_CRATE_100:
                  return new SimpleObject(
                        x, y, new Rectangle(100, 100), Material.IRON,
                        EObjectType.ITEM,
                        ImageLoader.getImage("metal_crate_100"));
```

Třída *ItemObjectFactory* funguje podobně jako *SimpleObjectFactory*, ale s tím rozdílem, že vytváří objekty, které může hráč sbírat do inventáře a popř. je použít. Podobně funguje také třída *VisibleStaticObjectFactory*, která ale na rozdíl od předešlých vytváří objekty statické (statické pevné těleso je popsáno v kapitole 3.7).

Poslední třída *ScenarioMaker* je speciální továrnou, která vytváří všechny možné typy objektů na začátku při inicializaci herní úrovně. Vytváří objekty potřebné pro scénář dané úrovně a využívá k tomu právě všech výše zmíněných továren.

4.4.3 Obecné

Třída *Material* definuje různé materiály využité ve hře. Při vytváření pevného tělesa v simulaci, mu lze nastavit doplňující vlastnosti. Díky třídě *Material* jsou tyto vlastnosti již přednastaveny a pouze je stačí přiřadit vytvářenému pevnému tělesu. Třída definuje materiály jako železo, dřevo, sklo, guma, hliník, apod.

Třída *ZoneController* má ve hře na starosti záchytné body. Při inicializaci herní úrovně jsou z textového souboru načteny jednotlivé body a tato třída hlídá to, jestli hráč daným bodem prošel. Pokud hráč bodem projde, nastaví se automaticky tento bod jako záchytný. Když potom hráč potřebuje resetovat pozici robota, objeví se právě v záchytném bodu. Tyto záchytné body jsou ještě rozděleny na vstup, záchytný bod a výstup. Na vstupu se hráč objeví při vstupu do herní úrovně a odchází z ní, pokud dosáhne výstupního bodu. Definice bodů vypadá následovně:

```
# Entrance
ENT,250,720,200,200
# Checkpoints
CHP,1317,641,200,200
CHP,2664,517,200,200
# Exit
EXI,3812,764,200,200
```

První tři písmena definují typ záchytného bodu, následují souřadnice X, Y a poslední dva parametry značí šířku a výšku aktivní zóny, která reprezentuje záchytný bod.

Další důležitou třídou je *Render*. V této třídě je popsáno, jak se mají vykreslovat jednotlivé herní objekty. Každý herní objekt, jak bude popsáno později, má atribut typu *Body*, tedy pevné těleso fyzického simulátoru. Toto pevné těleso se může skládat z různých tvarů (čtverec, kruh, apod.) a je tedy nutno tyto tvary vykreslovat zvlášť. Pro každý tvar proto existuje vlastní metoda na vykreslení jak je vidět v následujícím kusu kódu:

```
private static void drawShape(Graphics g, Body body, Shape shape) {
      if (shape instanceof Rectangle) {
            drawRectangle(g, body, (Rectangle) shape);
      }
      if (shape instanceof Circle) {
            drawCircle(g, body, (Circle) shape);
      }
      if (shape instanceof Polygon) {
            drawPolygon(g, body, (Polygon) shape);
      }
      if (shape instanceof CompoundShape) {
            drawCompound(g, body, (CompoundShape) shape);
      }
}
```

Neméně důležitými třídami jsou třídy *Loader*, *LoaderImage*, *LoaderSound* a *LoaderMusic*. Třída *Loader* je využita při prvotní inicializaci herních dat, kvůli správnému adresování souborů. Mimo jiné obsahuje například tuto metodu:

```
public static File getFile(String ref) {
      try {
            return new File(
                  ClassLoader.getSystemResource(ref).toURI().getPath());
      } catch (URISyntaxException ex) {
            return null;
      }
}
```

Třídy *LoaderImage*, *LoaderSound* a *LoaderMusic* obsahují kolekci *Map*, ve které uchovávají herní data připravené k okamžitému použití. Třída *LoaderImage* vypadá následujícím způsobem:

```
private static HashMap<String, Image> images = new HashMap<String,
Image>();

public static void load(String name, Image image) {
      images.put(name, image);
}

public static Image getImage(String name) {
      return images.get(name);
}
```

Ke všem herním datům se přistupuje pomocí názvů souborů bez přípony.

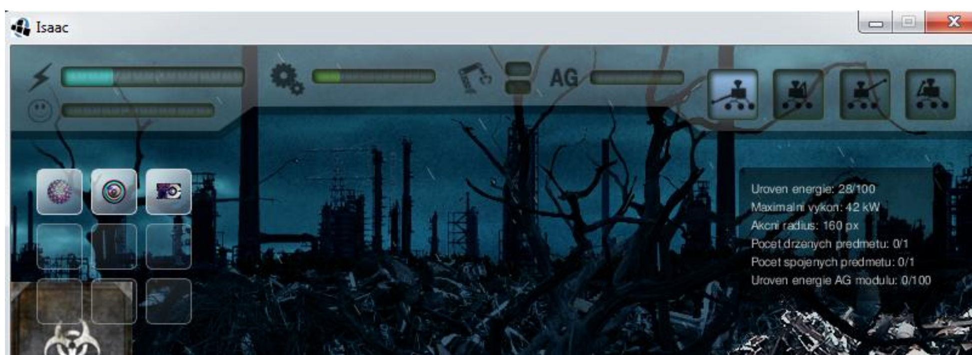
Jedna z nejdůležitějších tříd je *MarkedObject*. Tato třída pracuje se statickými atributy a metodami. Zajišťuje označení herního objektu, na který hráč najede myší a dále možnost jeho uzamknutí, aby zůstal objekt označený, i když hráč myší odjede pryč. Tato třída se tedy stává klíčovou pro interakci se všemi herními objekty. Pro označení objektu používá třída následující metodu:

```
public static void setLevelObject(ILevelObject o) {
     if (!isLocked) {
           markedObject = o;
           isMarked = true;
     }
}
```

4.4.4 Průhledový displej

Průhledový displej je součástí většiny her a zobrazuje hráči důležité informace na herní obrazovce. Třída, která se o tento průhledový displej stará, se jmenuje jednoduše *HUD*. Třída zobrazuje hráči graficky stav baterie robota Isaaca, aktuální výkon, stav robotické ruky (které z jejích částí jsou zapnuté nebo vypnuté), zvolenou pozici robotické ruky, úroveň důvěry u místních obyvatel planety a stav energie antigravitačního pohonu.

Třída *CharacterInformation* zajišťuje zobrazení hráči další údaje v číselné podobě. Mezi údaji lze nalézt stav baterie, maximální výkon, stav energie antigravitačního pohonu, maximální počet držených objektů, maximální počet spojených objektů a v neposlední řadě také seznam aktivních úkolů. Seznam aktivních úkolů se zobrazuje pouze v případě, když je alespoň jeden úkol aktivní.



Obrázek 5 – Průhledový displej ve hře

4.4.5 Další nástroje

Program využívá různé mini nástroje a mezi ty hlavní patří například třída *TextFileReader*, která slouží k načtení textového souboru. Jednotlivé řádky souboru načte do kolekce *ArrayList* a následně poskytne iterátor, kterým lze postupně řádky procházet a zpracovat dalšími algoritmy. Načtení jednotlivých řádků vypadá následovně:

```
InputStream in = Loader.getInputStream(path);
input = new BufferedReader(new InputStreamReader(in));
try {
      String s;
      while ((s = input.readLine()) != null) {
            if (!s.contains("#") && !s.isEmpty()) {
```

```
                buffer.add(s);
            }
        }
} catch (IOException ex) {}
```

Všechny řádky, které jsou prázdné nebo obsahují znak '#' jsou ignorovány, takže v textovém souboru mohou být komentáře.

4.5 Herní objekty

Herních objektů je celá řada. Jsou to struktury, které nesou klíčové herní informace nezbytné pro správné fungování celé hry.

V kapitole 3.5 bylo zmíněno, že objekt typu *Body* může nést uživatelem definovaná data. V této aplikaci je to objekt typu *DataObject*. Obsahuje informace o typu objektu a obrázek nebo animaci reprezentující herní objekt. Typ objektu je definován výčtovým typem *EObjectType*. Obrázek nebo animace je využita při vykreslování herního objektu.

4.5.1 Třídy StaticObject a DynamicObject

Základním kamenem herních objektů je buď třída *DynamicObject* nebo *StaticObject*. Obě dvě třídy implementují rozhraní *ILevelObject*, které předepisuje stěžejní metody. Rozhraní ILevelObject definuje následující metody:

```
void update(int delta);
void render(Graphics g);
boolean isAlive();
void notifyDeath();
Body getBody();
float getWorldX();
float getWorldY();
float getWidth();
float getHeight();
float getLocalCenterX();
float getLocalCenterY();
```

Rozdíl mezi těmito dvěma třídami je patrný z názvu. Třída *StaticObject* dovoluje vytvářet statické pevné těleso, zatím co třída *DynamicObject* dovoluje vytvářet dynamické pevné těleso.

Obě dvě třídy obsahují atribut typu *Body<IDataObject>*, tedy pevné těleso s definovanými uživatelskými daty. Dále obsahují atribut typu *Material*, který definuje vlastnosti pevného tělesa.

Třída *StaticObject* je rodičem tříd *InvisibleStatisObject* a *VisibleStaticObject*. Neviditelné statické objekty jsou využity k vytvoření základního koridoru, který je překryt hlavní texturou herní úrovně. Viditelné statické objekty jsou využity k vytvoření různých úchytů, ke kterým je možné přidělat jiné herní objekty.

Třída *DynamicObject* je rodičem mnoha tříd, mezi které patří například třída *SimpleObject*. Pomocí této třídy se vytváří základní jednoduché herní objekty. Tuto třídu hojně využívá továrna *SimpleObjectFactory* (popsáno v kapitole 4.4.2).

4.5.2 Třída ItemObject

Abstraktní třída *ItemObject* je potomkem třídy *DynamicObject* a implementuje rozhraní *IInventoryItem*. Chová se tedy jako každý další herní objekt s tím rozdílem, že lze tento objekt umístit do inventáře a tím ho odebrat z herní úrovně. Prázdná těla abstraktních metod implementují potomci této třídy, kteří utvářejí klíčové předměty hry. Tyto předměty se po sebrání zobrazí v inventáři hráče a lze je použít. Může se jednat o energetické články zvyšující úroveň energie baterie robota nebo úroveň energie antigravitačního modulu. Tyto předměty jsou vytvářeny pomocí tříd:

- *AntigravityModulItemObject* – umožňuje využití antigravitačního modulu,
- *AttachModulItemObject* – zvyšuje počet spojených objektu o 1,
- *BatteryItemObject* – zvyšuje úroveň energie baterií,
- *HoldModulItemObject* – zvyšuje počet držených předmětů o 1,
- *JetpackBatteryItemObject* – zvyšuje úroveň energie antigravitačního modulu,
- a *PowerModulItemObject* – zvyšuje maximální výkon.

4.6 Hráč

Hráč je klíčovým objektem ve hře a je reprezentovaný třídou *Player*, která implementuje rozhraní *MouseListener* a *KeyListener*, díky kterým může hráč ovládat celou hru.

Třída *Player* obsahuje atributy:

- String name – jméno hráče,
- Robot robot – herní objekt, který hráč ovládá (popsáno v kapitole 4.6.1),
- Inventory inventory – inventář na sebrané objekty (popsáno v kapitole 4.6.2),
- HUD hud, CharacterInformation character – grafické informace o herním objektu (popsáno v kapitole 4.4.4),
- double popularity – úroveň důvěry místních obyvatel planety, kterou si získává tím, že plní úkoly, které mu zadávají,
- a QuestList questList – seznam aktivních úkolů (popsáno v kapitole 4.6.3).

Kompletní ovládání je popsáno v kapitole 5.2.

4.6.1 Robot Isaac

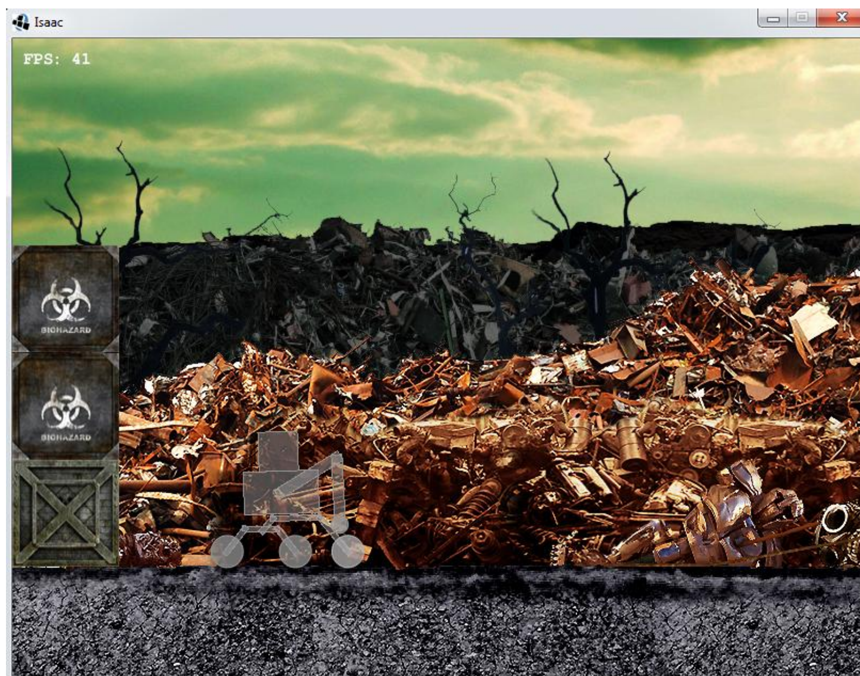
Třída *Robot* je nejrozsáhlejší třídou v celé aplikaci. Reprezentuje hlavní herní objekt – robota Isaaca. Třída obsahuje mnoho atributů, které uchovávají často měněné hodnoty – jako úroveň energie v bateriích, úroveň energie v antigravitačním pohonu – nebo méně proměnlivé hodnoty – jako maximální výkon, maximální počet spojených objektů, apod.

Robot se skládá z více částí, které jsou reprezentovány třídou *RobotPart*. Tato třída je potomkem třídy *DynamicObject* a definuje jeden atribut navíc. Tím je míra opotřebení jednotlivých dílů (bohužel v samotné hře se tento atribut nakonec nevyužil). Jednotlivé díly jsou uchovávány v kolekci *Map*, je jich celkem 14 a každý má svůj název. Vytvoření těla robota na souřadnicích X, Y ukazuje následující kus kódu:

```
IRobotPart body = new RobotPart(x, y + 37, new Rectangle(70, 40),
Material.ALUMINIUM, ImageLoader.getImage("isaac_body"), 0.5f);
```

Když jsou vytvořeny všechny díly robota, přejde se k dalšímu kroku a tím je vytvoření spojení mezi jednotlivými díly. V následujícím kusu kódu je ukázka vytvoření spojení mezi podvozkem a kolem:

```
joints.put("supportLeftAuxToRearWheel", JointFactory.createRevoluteJoint(
      supportLeftAux.getBody().getJBoxBody(),
      rearWheel.getBody().getJBoxBody(),
      new Vec2(rearWheel.getLocalCenterX() * World.METERS_PER_PIXEL,
      rearWheel.getLocalCenterY() * World.METERS_PER_PIXEL)));
```



Obrázek 6 – První koncept robota

Mezi základní funkce metod třídy *Robot* patří zjišťování různých souřadnic – levý horní roh pomyslného čtverce kolem robota, pravý dolní roh pomyslného čtverce kolem robota a absolutní střed pomyslného čtverce robota. Tyto souřadnice jsou zjišťovány algoritmy, které postupně prochází všechny díly robota a porovnávají jejich souřadnice:

```
public Vector2 getUpperLeft() {
      float x = Float.MAX_VALUE;
      float y = Float.MAX_VALUE;
      for (Map.Entry<String, IRobotPart> entry : parts.entrySet()) {
            if (x > entry.getValue().getWorldX()) {
                  x = entry.getValue().getWorldX();
            }
            if (y > entry.getValue().getWorldY()) {
                  y = entry.getValue().getWorldY();
            }
      }
      return new Vector2(x, y);
}
```

Díky těmto souřadnicím je pak velice jednoduché určit šířku nebo výšku robota:

```
public float getWidth() {
      return (getLowerRight().x - getUpperLeft().x);
}
```

Robot se pohybuje dopředu a dozadu pomocí svých kol, na které je vytvářen točivý moment síly. Ten se nastavuje pomocí metody, která hodnotu zapíše do proměnné a zároveň zmenší tření brzd na minimum:

```
public void setWheelTorque(double torque) {
      if (energy > 0) {
            this.torque = (float) torque;
      } else {
            this.torque = 0;
      }
      brakesFriction = 0.0001f;
}
```

Pokud chce robot zastavit, musí použít brzdy. Ty se aplikují tak, že se jim nastaví velké třetí, ale pouze za předpokladu, že robot nevyčerpal všechnu svoji energii:

```
if (energy > 0) {
      brakesFriction = 0.95f;
}
this.torque = 0;
```

Pro případ, že by se někde hráči robot zasekl, nebo by zcela vyčerpal energii, je tu možnost resetování – robot se objeví na posledním záchytném bodu. Toto obnovení je vyřešeno pomocí opětovné inicializace na určitých souřadnicích:

```
public void reinit(float x, float y) {
      parts.clear();
      releaseAllObjects();
      joints.clear();
      init(x, y);
      if (energy <= 0) {
            chargeBatteries(150000);
      }
}
```

Jednou z klíčových vlastností robota je možnost uchopit objekty. Maximální počet držených objektů je stanoven, ale lze ho během hry postupně navyšovat. K uchopení objektu je potřeba, aby byl objekt označený pomocí třídy *MarkedObject* (popsáno v kapitole 4.4.3) – díky tomu má program k dispozici instanci označeného objektu. V prvním kroku algoritmus zkontroluje, zdali již není vyčerpán maximální počet držených objektů. Pokud ano, odhodí robot objekt (odstraní spojení), který uchopil jako první. Ve druhém kroku je kontrolována vzdálenost robotovy paže od objektu. Pokud je objekt dostatečně blízko (v akční zóně) může být vytvořeno spojení. Prvním bodem, který je potřeba pro vytvoření spojení, je střed konce robotovy paže a druhým bodem je aktuální pozice kurzoru myši jak je vidět v následujícím kódu:

```
public void holdObject(ILevelObject o) { ...
```

```
        Vec2 a = new Vec2(getArmAnchor().x * World.METERS_PER_PIXEL,
        getArmAnchor().y * World.METERS_PER_PIXEL);
        Vec2 b = new Vec2(
              GameplayState.getGameController().getActiveLevel().
                    getCamera().ScreenXToWorldX(
                    input.getMouseX()) * World.METERS_PER_PIXEL,
                    GameplayState.getGameController().getActiveLevel().
                          getCamera().ScreenYToWorldY(input.getMouseY()) *
                          World.METERS_PER_PIXEL);
        DistanceJoint dj = JointFactory.createDistanceJoint(
              parts.get("handCraneBall").getBody().getJBoxBody(),
        o.getBody().getJBoxBody(), a, b);
... }
```

Odhození předmětu je realizováno pomocí metody, jejímž vstupním argumentem je objekt typu *ILevelObject*. V seznamu uchopených objektů je vyhledána shoda a objekt je odstraněn. Robot může také odhodit všechny objekty najednou tak, že jsou postupně v cyklu odstraňovány všechny spojení.

```
public void releaseObject(ILevelObject o) {
      GameplayState.getGameController().getActiveLevel().getWorld().
            getJBoxWorld().destroyJoint(holdObjects.get(o));
      holdObjects.remove(o);
}
```

Při uchopení objektu existuje jistá vzdálenost mezi paží robota a označeným objektem. Tato vzdálenost se může v určitých mezích měnit – zkracovat nebo prodlužovat. Podmínka jestli je možné provést změnu vzdálenosti je poněkud složitá a bez hlubší znalosti knihovny JBox2D je bezpředmětné jí zde uvádět. Nicméně metoda, která zajišťuje změnu vzdálenosti, obsahuje kromě zmíněné složité podmínky následující kusy kódu:

```
public void changeDistanceOfHoldObject(ILevelObject o, double dist) {
      if (!holdObjects.containsKey(o)) {
            return;
      }
      if (...) {
            ((DistanceJoint) holdObjects.get(o)).m_length += dist;
      }
}
```

Mimo výše zmíněné metody ještě obsahuje třída metodu, která kontroluje, zdali je označený objekt již uchopen:

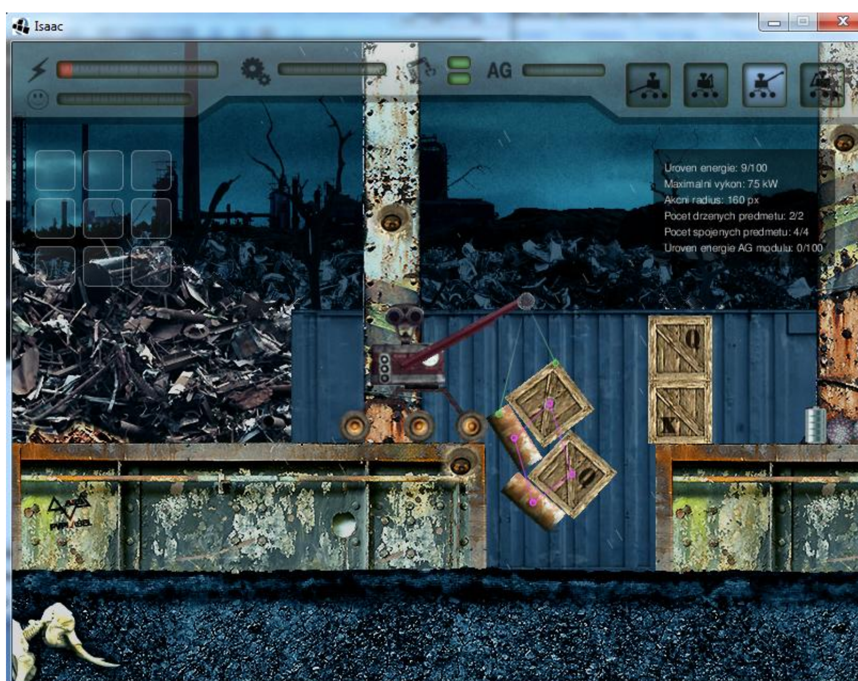
```
public boolean isObjectAlreadyHold(ILevelObject o) {
      return holdObjects.containsKey(o);
}
```

Další klíčová vlastnost robota je možnost spojování objektů k sobě. To zajišťuje metoda, která jako své vstupní argumenty požaduje pár objektů, které k sobě mají být spojeny. Prvním krokem algoritmu je kontrola, zdali není vyčerpán maximální počet spojených objektů. Druhým krokem algoritmu, za předpokladu že je první podmínka splněna, je vytvoření dvou bodů, kde každý reprezentuje střed jednoho objektu. Následně je vytvořeno

spojení, které je zároveň uloženo do kolekce pro pozdější operace. Kus metody je v následujícím příkladu:

```
public void attachObjects(Pair<ILevelObject> objects) { ...
      ILevelObject o1 = objects.getFirst();
      ILevelObject o2 = objects.getSecond();
      DistanceJoint joint = JointFactory.createDistanceJoint(
            o1.getBody().getJBoxBody(), o2.getBody().getJBoxBody(),
            new Vec2(o1.getLocalCenterX() * World.METERS_PER_PIXEL,
            o1.getLocalCenterY() * World.METERS_PER_PIXEL),
            new Vec2(o2.getLocalCenterX() * World.METERS_PER_PIXEL,
            o2.getLocalCenterY() * World.METERS_PER_PIXEL));
... }
```

Odstranění spojení mezi objekty je obdobné jako při odstraňování uchopených objektů. V cyklu je hledána shoda s označeným objektem, který je následně odstraněn a spojení je zničeno.



Obrázek 7 – Manipulace s robotickým ramenem

Další vlastností robota je možnost házet objekty v určitém směru. Pokud je označen předmět pomocí třídy *MarkedObject* a hráč táhne kurzorem myši směrem od objektu, je mu vykreslena síla a směr hodu. Tyto hodnoty jsou potom předány jako parametry metodě, která se stará o to, že označenému objektu aplikuje impuls. Zároveň je pomocí čítače zajištěno, aby nešlo aplikovat impuls vícenásobně za sebou a tím hodit objekt nesmyslně velkou silou.

```
public void throwObject(ILevelObject o, Vector2 direction, float force) {
      if (action_throw_countdown.finished()) {
            if (isObjectInActionZone(o) && o != null) {
                  o.getBody().applyImpulse(
                        direction.x * force, direction.y * force);
            }
```

```
            action_throw_countdown.reset();
      }
}
```

Výpočet aktuálního výkonu zajišťuje metoda, která do výpočtu zahrnuje velikost točivého momentu každého kola, používání robotické paže a spotřebu antigravitačního modulu. Algoritmus se snaží částečně korespondovat s realitou, ale spíše je vyladěný pro optimální využití ve hře.

V předešlých metodách je na pár místech využita metoda, která kontroluje, jestli je označený objekt v akční zóně, tedy jestli s ním může robot pracovat. Využívá k tomu funkci pro výpočet vzdálenosti mezi dvěma body, která je ve třídě *MathAlg*:

```
public static double getDistance(Vector2 a, Vector2 b) {
      return (Math.sqrt(
      Math.pow((a.x - b.x), 2) + Math.pow((a.y – b.y), 2)));
}
```

Metoda pro kontrolu akční zóny vypadá následovně:

```
public boolean isObjectInActionZone(ILevelObject o) {
      if (o == null) {
            return false;
      }
      double dist = MathAlg.getDistance(getArmAnchor(),
            new Vector2(o.getLocalCenterX(), o.getLocalCenterY()));
      if (dist <= MAX_DISTANCE_FOR_ACTION) {
            return true;
      }
      return false;
}
```

Ovládání robotické paže je realizováno pomocí několika metod. Tyto metody zajišťují změnu nastavení spojení, které realizuje otočné klouby robotické paže. Jedná se o spojení definované třídou *RevoluteJoint*, u které je možné nastavit maximální a minimální úhel otočení. Další z možností je zapnutí a vypnutí tzv. motoru, který otáčí kloubem ve směru, který lze určit nastavením rychlosti. Rychlost může mít jakoukoli velikost a také může nabývat kladných i záporných hodnot. Zapnutí motoru může vypadat následovně:

```
((RevoluteJoint) joints.get("handCranePart1ToBody")).enableMotor(true);
```

Nastavení rychlosti a směru otáčení vypadá takto:

```
((RevoluteJoint)joints.get("handCranePart1ToBody")).setMotorSpeed((float)
Math.toRadians(speed));
```

V některých kusech kódu byla zmíněna metoda, která poskytuje souřadnice středu konce robotické paže:

```
public Vector2 getArmAnchor() {
      return new Vector2(parts.get("handCraneBall").getLocalCenterX(),
            parts.get("handCraneBall").getLocalCenterY());
}
```

Třída *Robot* obsahuje ještě několik dalších metod, které zajišťují poskytování nebo nastavování určitých hodnot. Kromě těchto metod jsou tu ještě dvě poslední velice důležité metody a těmi jsou metoda pro vykreslování a metoda pro aktualizaci.

Metoda pro aktualizaci ve svém těle obsahuje hlavně kód, který se stará o pohybovou složku robota. V kódu také nalezneme kousek, který se stará o aktualizaci úrovně energie robota:

```
energy -= Math.abs(getActualPower()) * 0.0066f;
      if (energy < 0) {
            energy = 0;
      }
```

Tento kousek kódu se provede, pokud je úroveň energie větší než nula. V opačném případě vypne všechny systémy robota a nechá ho na pospas bez možností dalšího ovládání.

Pokud je nastaven točivý moment kol (anglicky torque), aplikuje se při každé aktualizaci na objekty, které reprezentují kola robota:

```
if (getActualPower() < MAX_POWER) {
      parts.get("rearWheel").getBody().applyTorque(torque);
      parts.get("frontWheel").getBody().applyTorque(torque);
      parts.get("midWheel").getBody().applyTorque(torque);
}
```

Již bylo zmíněno, že robot může použít brzdy, tím že se nastaví velikost tření. Hodnota tření (proměnná *brakesFriction*) je použita při aktualizaci následujícím způsobem:

```
parts.get("rearWheel").getBody().setAngularVelocity(
      parts.get("rearWheel").getBody().getAngularVelocity() * (1 -
            brakesFriction));
```

Metoda pro vykreslování zajišťuje vykreslení všech částí robota, včetně různých spojení. Jako první se v cyklu vykreslují spojení mezi jednotlivými díly robota téměř průhlednou bílou barvou. Následuje vykreslení spojení mezi objekty, které robot drží – tyto čáry jsou vykresleny zelenou barvou. V dalším cyklu se vykreslují spojení mezi objekty, které byly spojeny robotem – tyto čáry jsou světle fialovou barvou. Vykreslení obrázků, které reprezentují jednotlivé díly robota, probíhá úplně v jiné třídě (popsáno v kapitole 4.7).

4.6.2 Inventář

Třída *Inventory* reprezentuje úložný prostor robota, ve kterém může uchovávat předměty a využít je až později. Úložný prostor má určitou kapacitu, která je definována při vytváření instance třídy. Mezi další atributy patří seznam předmětů, které jsou uloženy v kolekci *ArrayList* a objekt typu *IInventoryItem*, který v sobě uchovává objekt, nad kterým je právě kurzor myši.

Objekt typu *ILevelObject*, který je vložen do inventáře, je automaticky odebrán z herní úrovně a existuje pouze v inventáři jako objekt typu *IInventoryItem*.

Jednotlivé položky inventáře jsou v pomyslných čtvercích, které jsou počítány pomocí metody, která vypadá následovně:

```
private Rectangle getItemRect(int index) {
	int i = (int) (Math.floor(index / ITEMS_IN_ROW) + 1);
	int j = (int) (index - (i - 1) * ITEMS_IN_ROW + 1);
	return new Rectangle((position.x + (j - 1) * 45), (position.y + (i
		- 1) * 45), 40, 40);
}
```

Proměnná *ITEMS_IN_ROW* určuje kolik předmětu je zobrazeno na jednom řádku a proměnná *position* určuje souřadnice, na kterých začíná vykreslení grafické podoby inventáře.

Obrázek 8 – Grafická podoba inventáře

Při pohybu kurzoru myši nad inventářem se opakovaně volá metoda, pomocí které je následně nastaven předmět, nad kterým je kurzor.

```
private int getMouseOverItemIndex() {
	for (int i = 0; i < items.size(); i++) {
		if (getItemRect(i).contains(input.getMouseX(),
				input.getMouseY())) {
			return i;
		}
	}
	return -1;
}
```

4.6.3 Úkoly

Jednotlivé úkoly jsou reprezentovány třídou *Quest*, pomocí které se definují podmínky splnění úkolu a název. Úkol je splnitelný ve chvíli, kdy má robot ve svém inventáři předmět, který má určité identifikační číslo. Teprve po té co přijde k NPC postavě, která mu úkol zadala, je úkol splněn. Metoda, která zjišťuje, jestli je úkol splnitelný, vypadá následovně:

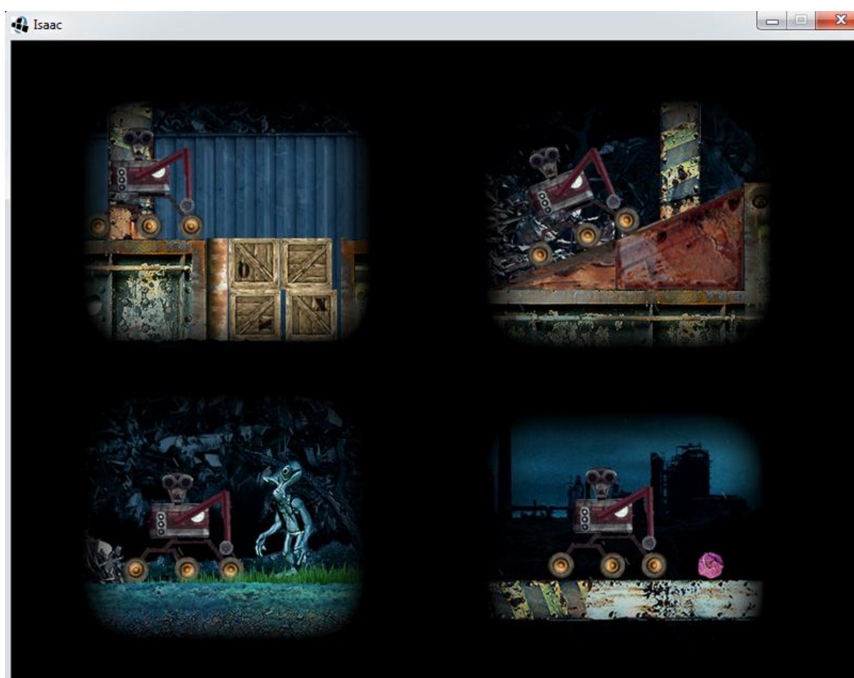
```
public boolean isCompletable() {
	Iterator it = GameplayState.getGameController().getPlayer().
		getInventory().getIterator();
	while (it.hasNext()) {
		if (it.next().hashCode() == questItemHashCode) {
```

```
                return true;
            }
        }
        return false;
}
```

Označení úkolu jako splněný je realizováno zavoláním následující metody:

```
public void notifyComplete() {
        complete = true;
}
```

Všechny úkoly jsou uchovávány ve třídě *Player* v atributu typu *QuestList*. Třída *QuestList* je pouze kontejner, který ukládá úkoly do kolekce *Map*, kde klíčem je identifikační číslo úkolu a hodnota je samotná instance úkolu.



Obrázek 9 – Návod na skryté obrazovce ve hře

4.7 Herní úroveň

Třída *Level* reprezentuje herní úroveň. Obsahuje mnoho atributů, mezi které patří například objekt typu *World*, který je hlavní spojkou mezi hrou a fyzikálním simulátorem. Dalším důležitým atributem je seznam všech herní objektů, které jsou uchovávány v kolekci *ArrayList*.

Mezi atributy třídy *Level* je objekt typu *Camera*. Třída *Camera* se stará o posouvání herní obrazovky a sledování robota, tak aby byl stále uprostřed (pokud je to možné).

Horizont a celé pozadí herní úrovně má na starosti třída *LevelBackround*, která v určitém poměru vykresluje jednotlivé vrstvy pozadí, tak aby se pohybovaly věrohodně.

Při inicializaci samotné herní úrovně je vytvořen objekt třídy *LevelLoader*, který zajišťuje přečtení textového souboru, ve kterém jsou definovány základní herní objekty. Tyto objekty po načtení souboru vytvoří a automaticky je přidá do herní úrovně. Následující ukázka je z výše zmiňovaného textového souboru, kde jednotlivé parametry jsou popsány v hlavičce originálního zdrojového souboru:

```
SI 0,896,1049,103,0.0,r,0,stone
SI 1047,775,451,118,0.0,r,0,metal
```

Třída *Level* dále obsahuje metodu pro vykreslování a metodu pro aktualizaci. V metodě pro vykreslování je postupně vykresleno pozadí, speciální efekty jako počasí (popsáno v následující kapitole 4.8), všechny obrázky a animace spojené s herními objekty a nakonec popředí. V metodě pro aktualizaci je postupně aktualizován objekt, který se stará o záchytné body, objekt reprezentující hráče, fyzikální simulátor, herní objekty a nakonec speciální efekty.

V této třídě také probíhá označování herního objektu, který je následně uchován ve třídě *MarkedObject*. Při pohybu kurzoru myši se opakovaně volá metoda, která kontroluje zdali nejsou souřadnice myši uvnitř pomyslného čtverce reprezentujícího herní objekt. Algoritmus automaticky přeskakuje herní objekty, které jsou instance tříd *RobotPart*, *NPC* a *InvisibleStaticObject*, aby nemohlo dojít k nežádoucímu stavu.

4.8 Efekty

Obrazové efekty jsou pouze experimentálního charakteru. Třída *WeatherSystem* pracuje s rozhraním *IWeather*, které definuje tři metody – inicializaci, aktualizaci a vykreslování. Toto rozhraní implementuje třída *Rain*, která zajišťuje v herním světě efekt deště. Dokola generuje dešťové kapky, které jsou reprezentovány vnitřní třídou *Drop*. Tato vnitřní třída má dva atributy – souřadnice X, Y a délku čáry. Tyto kapky jsou uchovávány v kolekci *ArrayList*, ze které jsou po dopadu na zem odebrány a nahrazeny novou další kapkou, která padá z vrchu.

Zvukové efekty se snaží navodit správnou atmosféru hry. Spousta úkonů, které robot dělá, je doprovázeno nějakým zvukem. Přehrávání zvuku, s určitou základní výškou tónu a hlasitostí, se realizuje následujícím způsobem:

```
SoundLoader.getSound("robot_beep").play(1, 0.5f);
```

Podobným způsobem se přehrává i hudba, která dokresluje celou atmosféru hry. Všechny zvuky pocházejí ze zvukových bank, které poskytují zvuky zdarma ke svobodnému užití. Autorem dvou použitých skladeb jsem já sám osobně.

5 Ovládání aplikace

5.1 Spuštění aplikace

Aplikace je distribuována v archivu ZIP. Po rozbalení archivu do jakéhokoli umístění, lze v hlavní kořenové složce nalézt soubor *run.bat*. Tento soubor obsahuje parametry nutné ke správnému spuštění aplikace.

5.2 Klávesnice a myš

Ovládání hry je popsáno v nápovědě pod klávesou F1 v herním režimu. Samotná implementace ovládání obsahuje velice mnoho řádků kódu a je možné jí najít ve zdrojových kódech ve třídě *Player*. Ovládání robota je popsáno v následujících tabulkách (Tabulka 1 – Funkční klávesy a Tabulka 2 – Funkční tlačítka myši).

Tabulka 1 – Funkční klávesy

Klávesa	Funkce
A	Jízda / let vlevo
D	Jízda / let vpravo
W	Skok
S	Zabrzdění
Q	Aktivace prvního kloubu robotické paže
E	Aktivace druhého kloubu robotické paže
1	Nastavení robotické paže do polohy 1
2	Nastavení robotické paže do polohy 2
3	Nastavení robotické paže do polohy 3
4	Nastavení robotické paže do polohy 4
5	Zastavení robotické paže
I	Zobrazení/nezobrazení inventáře
C	Zobrazení podrobných informací o robotovi
P	Obnovení robota na posledním záchytném bodu (mírné dobytí energie baterií, pokud jsou zcela vybité)
Mezerník	Aktivace antigravitačního modulu (přidržení)
ESC	Pozastavení hry
F1	Nápověda
F2 + F3	Obrázkový návod zobrazující řešení klíčových herních hádanek

Tabulka 2 – Funkční tlačítka myši

Tlačítko	**Akce**	**Funkce**
Pravé tlačítko	klik	Sbírání předmětů do inventáře
	dvojklik	Použití předmětů (v inventáři)
	tažení	Házení objektů
Prostřední tlačítko	tažení	Spojování objektů
	klik	Rozpojování objektů
Levé tlačítko	klik	Uchopení / odhození objektu
Kolečko	rolování	Změna délky uchopeného objektu

Závěr

Tato práce je věnována vývoji 2D logické hry pomocí programovacího jazyka Java. K vývoji hry byly použity dvě hlavní externí knihovny. První knihovna, která se jmenuje Slick2D, zajišťuje vývojové nástroje pro vytvoření hry a je částečnou nadstavbou nad knihovnou Lightweight Java Game Library. Ta využívá knihovny OpenGL k vykreslování grafiky a OpenAL k přehrávání zvuku. Knihovna Slick2D poskytuje základní koncept hry rozdělené na stavy a v jednotlivých stavech poskytuje programátorovi metody pro inicializaci, vykreslování a aktualizaci hry. Druhá klíčová knihovna se jmenuje JBox2D a jedná se o fyzikální simulátor, který pracuje se systémem KMS (kila – metry – sekundy). Tento simulátor je využit k realizaci herní fyziky a vytvoření interaktivního herního světa. Díky implementaci věrné fyziky se hra stává poměrně obtížnou výzvou pro hráče.

Hlavní postavou hry je malý robot, kterého hráč ovládá pomocí klávesnice a myši. Robot umí manipulovat s objekty, spojovat objekty mezi sebou nebo je odhazovat v určitém směru a určitou silou. Také obsahuje jakýsi úložný prostor, do kterého může ukládat předměty, které může využít později. Cílem hry je projít celou herní úroveň – dostat se přes všechny překážky – a splnit zadaný úkol, který robot dostane před koncem od jedné nehratelné postavy. Po celou dobu hry se hráč potýká s věrnou fyzikou, která tvoří významnou část všech hádanek. Hru doprovází zvukové efekty a hudební doprovod, který jsem sám skládal.

Celá hra je v rámci možností odladěna a měla by být bez větších problému hratelná až do samotného konce. Obtížnost je díky věrné fyzice z mého názoru poměrně vysoká, ale při opakovaném hraní by měl být postupem času každý hráč schopen projít hrou až do samého konce hratelné ukázky.

Hra byla vyvíjena s ohledem na možnost dalšího rozšiřování, zejména co se týče dalších herních úrovní a dalších herních objektů. Také je možné implementovat další grafické efekty počasí (např. sněžení), díky rozhraní, které definuje potřebné metody.

Literatura

[1] DAWISON, Andres. *Programování dokonalých her v Javě.* Vyd. 1. Překlad Lukáš Krejčí. Brno : Computer Press, 2004, 421 s. ISBN 80-722-6944-5.

[2] MORRISON, Michael. *Naučte se programovat počítačovou hru za 24 hodin*. Vyd. 1. Brno : Computer Press, 2004, 421 s. ISBN 80-251-0371-4.

[3] DARWIN, Ian F. *Java: kuchařka programátora.* Vyd. 1. Brno : Computer Press, 2006, 798 s. ISBN 80-251-0944-5.

[4] GLASS, Kevin. *Slick2D: Open Source 2D Java Game Library* [online]. 2012 [Citace: 14. 4. 2013]. http://www.slick2d.org/

[5] GLASS, Kevin. *Slick2D: Wiki* [online]. 2012, 29. 10. 2012 [Citace: 14. 4. 2013]. http://www.slick2d.org/wiki

[6] GLASS, Kevin. *Slick2D Wiki: Setting up Slick2D with NetbeansIDE* [online]. 25. 10. 2012. [Citace: 14. 4. 2012]. http://www.slick2d.org/wiki/index.php/Setting_up_Slick2D_with_NetBeansIDE

[7] LWJWL.ORG. *LWJGL: Lightweight Java Game Library* [online]. 2012 [Citace: 14. 4. 2013]. http://lwjgl.org/

[8] XIPH.ORG FOUNDATION. *The ogg container format* [online]. 2013 [Citace: 14. 4. 2013]. http://www.xiph.org/ogg/

[9] MURPHY, Daniel. *JBox2D: A Java Physics Engine* [online]. 2012 [Citace: 14. 4. 2013] http://www.jbox2d.org/

[10] CATTO, Erin. *Box2D: About* [online]. 2011 [Citace: 14. 4. 2013] http://box2d.org/about/

[11] CATTO, Erin. *Box2D v2.2.0 User Manual* [online]. 2011 [Citace: 14. 4. 2013] http://box2d.org/manual.pdf

[12] GLASS, Kevin. *Coke And Code: Libraries* [online]. 2013 [Citace: 23. 4. 2013] http://www.cokeandcode.com/index.html?page=libs

[13] WIKIPEDIE. *Garbage collector* [online]. 2013, 27. 3. 2013 [Citace: 1. 5. 2013] http://cs.wikipedia.org/wiki/Garbage_collector

[14] WIKIPEDIE. *Kodek* [online]. 2013, 16. 4. 2013 [Citace: 1. 5. 2013] http://cs.wikipedia.org/wiki/Kodek

Příloha A – Použití JBox2D

V této části je popsáno, jak vytvořit jednoduchou simulaci, ve které bude jedno dynamické pevné těleso, které bude padat a statické pevné těleso reprezentující zem.

Každá simulace začíná tím, že se vytvoří instance objektu typu *World*. Nejdříve vytvoříme vektor gravitace a proměnnou, která reprezentuje pravdivostní hodnotu, která stanovuje, jestli mohou tělesa v simulaci „spát" – tedy stát se neaktivní a tím šetřit výpočetní výkon.

```
Vec2 gravity = new Vec2(0.0f, -9.8f);
boolean doSleep = true;
```

Nyní můžeme vytvořit svět, ve kterém se bude odehrávat simulace.

```
World world = new World(gravity, doSleep);
```

Když máme vytvořený svět, můžeme přejít k definování a vytváření dalších objektů. Jako první vytvoříme objekt, který bude představovat zem.

```
BodyDef groundBodyDef = new BodyDef();
groundBodyDef.position.set(0.0f, -10.0f);
```

Nyní máme vytvořenou definici pevného tělesa a potřebujeme vytvořit těleso samotné. To uděláme pomocí továrny, kterou poskytuje třída *World*.

```
Body groundBody = world.createBody(groundBodyDef);
```

Jako další krok, který je potřeba k vytvoření objektu, který reprezentuje zem, musíme vytvořit tvar.

```
PolygonShape groundBox = new PolygonShape();
groundBox.setAsBox(50.0f, 10.0f);
```

Tím jsme nastavili objektu reprezentující zem 100 metrů na šířku a 20 metrů na výšku. Jako poslední krok k vytvoření země potřebujeme vytvořit vlastnosti objektu.

```
groundBody.createFixture(groundBox, 0.0f);
```

Tím máme vytvořenou zem. Objekt je implicitně statický, takže se nebude pohybovat a zůstane po celou dobu simulace na stejném místě.

Dynamický objekt reprezentující padající těleso vytvoříme stejným způsobem jako předcházející objekt reprezentující zem.

```
BodyDef bodyDef = new BodyDef();
bodyDef.type = BodyType.DYNAMIC;
bodyDef.position.set(0.0f, 4.0f);
Body body = world.createBody(bodyDef);
```

Dále vytvoříme tvar, který bude reprezentovat padající těleso.

```
PolygonShape dynamicBox = new PolygonShape();
dynamicBox.setAsBox(1.0f, 1.0f);
```

Když máme vytvořené tělo a tvar, přidáme definici vlastností.

```
FixtureDef fixtureDef = new FixtureDef();
fixtureDef.shape = dynamicBox;
fixtureDef.density = 1.0f;
fixtureDef.friction = 0.3f;
body.createFixture(fixtureDef);
```

Nyní jsme již kousek od samotné simulace, ale před tím, ještě musíme definovat časový krok, který může být například 1/60 sekundy. Dále ještě musíme definovat počet iterací na výpočet rychlosti a pozice objektů.

```
float timeStep = 1.0f / 60.0f;
int velocityIterations = 6;
int positionIterations = 2;
```

Když máme vše nastaveno, můžeme pustit simulaci a sledovat textový výstup.

```
for (int i = 0; i < 60; ++i) {
      world.step(timeStep, velocityIterations, positionIterations);
      Vec2 position = body.getPosition();
      float angle = body.getAngle();
      System.out.printf(
            "[%4.2f;%4.2f] %4.2f\n", position.x, position.y, angle);
}
```

Z textového výpisu je patrné, že padající těleso opravdu padá až nakonec dopadne na zem, kde zůstane až do konce simulace.

```
// [SOURADNICE_X;SOURADNICE_Y] UHEL_OTOCENI
[0,00;4,00] 0,00
[0,00;3,99] 0,00
[0,00;3,98] 0,00
[0,00;3,97] 0,00
[0,00;3,96] 0,00
[0,00;3,94] 0,00
[0,00;3,92] 0,00
[0,00;3,90] 0,00
...
[0,00;1,18] 0,00
[0,00;1,06] 0,00
[0,00;1,01] 0,00 // padající těleso dopadlo na zem
[0,00;1,01] 0,00
[0,00;1,01] 0,00
[0,00;1,01] 0,00 // konec simulace
```