Gymnázium, Praha 6, Arabská 14

Programování

ROČNÍKOVÁ PRÁCE

2019 Nguyen Tien, Grosman, Pelantová

Gymnázium, Praha 6, Arabská 14

Arabská 14, Praha 6, 160 00

ROČNÍKOVÁ PRÁCE

Předmět: Programování
Téma: RPG hra

Autoři: Nguyen Tien Dung, Lukáš Grosman, Michaela Pelantová
Třída: 3.E
Školní rok: 2018/2019
Třídní učitel: Mgr. Jana Urzová

Prohlašujeme, že jsme jedinými autory tohoto projektu, všechny citace jsou řádně označené a všechna použitá literatura a další zdroje jsou v práci uvedené. Tímto dle zákona 121/2000 Sb. (tzv. Autorský zákon) ve znění pozdějších předpisů udělujeme bezúplatně škole Gymnázium, Praha 6, Arabská 14 oprávnění k výkonu práva na rozmnožování díla (§ 13) a práva na sdělování díla veřejnosti (§ 18) na dobu časově neomezenou a bez omezení územního rozsahu.

V Praze dne _____

_____

vlastnoruční podpis autorů

ANOTACE:

Tato ročníková práce se zabývá tvorbou a problematikou multiplayerové hry žánru Role Playing Game – RPG (v překladu: hra na hrdiny). V práci jsou uvedeny technologie/programy, které jsme při práci na projektu použili. Dále je v práci popsána tvorba příslušných textur, samotný průběh programování této hry a jak jsme řešili konkrétní problémy, které při tvorbě hry nastaly.

ANNOTATION:

This thesis is about the creation and matters of multiplayer Role Playing Game – RPG. In our thesis we included the technologies/programs we used when working on the project. The thesis contains description of creating applicable textures, the process of programming and how we solved particular problems which have appeared while working on the project.

OBSAH:

1. ÚVOD

Toto téma jsme si vybrali, protože sami rádi hrajeme různé (nejen RPG) hry a chtěli jsme si zkusit, jaké to je takovou RPG hru vlastnoručně naprogramovat. Od této práce očekáváme, že se zlepšíme v programování, vytváření grafiky a týmové spolupráci.

Cílem hry je projít celou mapu a posbírat předměty potřebné k dokončení hry. Předměty se mohou nacházet kdekoliv na mapě, k některým se hráč musí probourat. Po sesbírání všech "klíčů" se hráči otevře portál, do kterého když se vkročí, ukončí hru.

2. Příběh

Ahoj!
Jmenuju se Evžen a nejsem si jistý, kde se právě nacházím. Poslední, co si pamatuju, je moment, kdy jsem doma otevíral novou knížku, co jsem si včera koupil. Vypadá to, že mě vtáhla do sebe a já se teď nějak musím dostat ven - zpátky do reálného světa.
Pomoz mi v tomto světě najít 3 magické klíče. Ty dohromady otevřou portál, který mě přenese zpět domů.
Klíče bys měl najít v bludištích, do kterých se dostaneš skrze brány. Vypadají poněkud staře, takže asi budeš muset použít trochu síly, abys je otevřel.
Kdybys náhodou cestou narazil na jakési ingoty, určitě je vem s sebou! Třeba se nám je podaří pronést skrz portál, abych je mohl doma prozkoumat (když už nic jiného, tak budu mít aspoň hezkej suvenýr...).
To dáme!

3. Použité technologie

Jako vývojové prostředí jsme zvolili Libgdx. Pro vytvoření grafické části hry byl využit program Piskel.

4. Libgdx

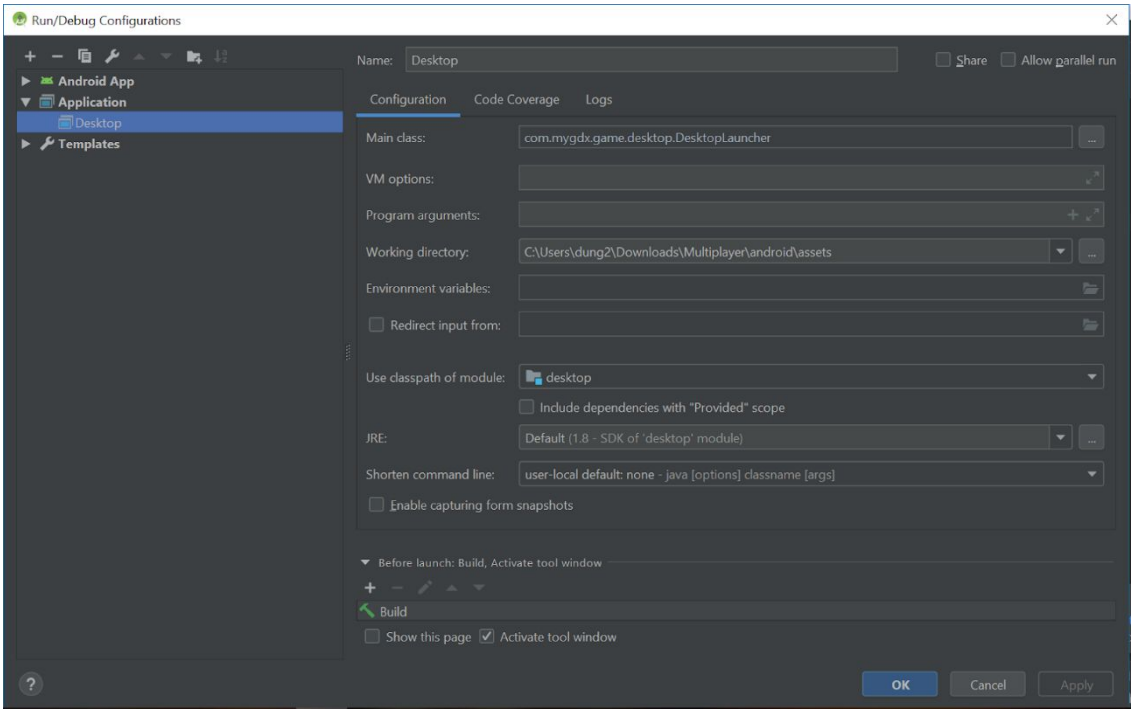
Libgdx umožňuje vývoj her na jakoukoliv platformu jako je Windows, Mac, Linux, Android, iOS, BlackBerry a HTML5 za použití programovacího jazyka Java. Je široce využíván a má velkou komunitu lidí. Má skvělou wiki stránku nebo javadoc, kde se může člověk rychle naučit jak Libgdx používat.

Po stáhnutí LibGDX z jejích oficiálních stránek jsme dostali panel pro generování projektů (viz obr. 1).

Obr. 1

Po nastavení různých požadavků jsme nechali generovat projekt. Projekt jsme poté otevřeli v Android Studiu. LibGDX generuje jakýsi svůj *„Hello World“* program, ale abychom ho mohli zkompilovat, museli jsme nejprve udělat menší konfiguraci. V okně

„Run/Debug Configurations“ jsme přidali novou konfiguraci *„Application“* a nastavili (viz obr. 2).



Obr. 2

5. Box2D

Knihovna Box2D nám zajišťuje práci s fyzikou hry. Tato knihovna vytváří jakýsi imaginární model hry, díky ní můžeme definovat vlastnosti těl v našem světě a také nám ulehčila práci s hitboxy.

Box2D obsahuje třídu *World* (česky svět), ve které se nachází instance třídy *Body* (česky tělo). Vlastnosti těl jsou následující. Jedna z vlastností těl je definovaná přes třídu *BodyDef*, kde je tělo určené buď jako dynamické, statické, nebo kinematické.

- ***Dynamické tělo*** - působí na něj síly a je ovlivněno ostatními těly. Dynamické tělo představuje tělo našeho hráče.
- ***Statické tělo*** - opak dynamického těla - nepůsobí na něj síly, není ovlivněno ostatními těly a nehýbe se. Statické tělo je tedy využito na modely budov a celkového terénu.
- ***Kinetické tělo*** - něco mezi statickým a dynamickým tělem. Stejně jako dynamické tělo se může hýbat, ale podobně jako statické na něj nepůsobí síly. V našem projektu jsme pro něj nenašli využití.

Tření těla, odpor vůči vzduchu a hustotu definuje třída *FixtureDef*. Aby se vlastnosti nemuseli definovat pro každé tělo zvlášť, vytvořili jsme třídu *BodyFactory*. Zde se nachází metody, které určují pouze parametry. Na třídu *BodyFactory* je použitý designový pattern singleton, jelikož jsme nechtěli více instancí té třídy a také jsme tím ušetřili paměť.

Náš modelový svět je vytvořený ve třídě *B2dModel*. Tady se přidávají těla do našeho modelového světa a řeší se zde také uživatelský vstup.

Pro vytváření mapy jsme použili aplikaci Tiled editor map, kde jsme si mohli načíst vlastnoručně vytvořené textury a zakomponovat je do mapy jako “tile”. Editor umožňuje nastavit “layeru” (vrstvě) *RectangleObject*, který pak jsme mohli načíst přes metodu *public void builBuildingsBodies()* (viz obr. 3.1) a tím vytvářet těla pro layery s hitboxem. Tak jsme se vyhnuli tomu, že bychom museli každý objekt zvlášť ručně vytvářet a jenom v Tiled nakreslíme chtěné “rectangly” (čtverce). Metoda *public void builBuildingsBodies()* projde všemi layery a pro každého z nich vytvoří “body” (tělo) a

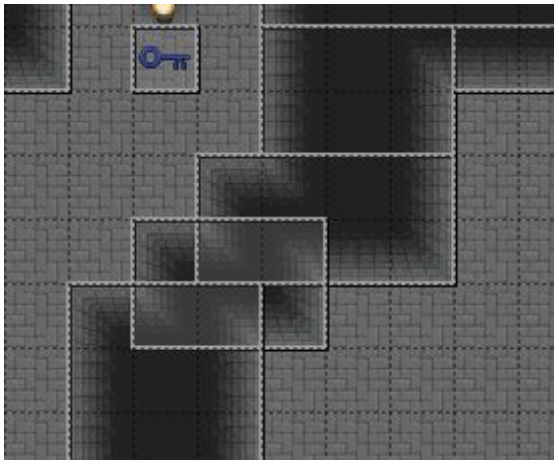
fixture (vlastnosti), metoda *public static Shape getShapeFromRectangle()* se stará o to,aby příslušné rectangly měly správné velikosti a udala mu tvar, jelikož soubory v Tiled jsou ukládány v pixelových rozměrech a Box2D zas ve svých rozměrech. Poslední metoda se stará o to, aby daná "body" byla umístěna na správných místech.

```
public void buildBuildingsBodies(TiledMap tiledMap, World world, String layer,String userData){
    MapObjects objects = tiledMap.getLayers().get(layer).getObjects();
    for (MapObject object: objects) {
        Rectangle rectangle = ((RectangleMapObject)object).getRectangle();
        BodyDef bodyDef = new BodyDef();
        bodyDef.type = BodyDef.BodyType.StaticBody;
        Body body = world.createBody(bodyDef);

        Fixture fixture = body.createFixture(getShapeFromRectangle(rectangle), density: 1);
        fixture.setUserData(userData);
        fixture.setFriction(0.1F);

        body.setTransform(getTransformedCenterForRectangle(rectangle), angle: 0);
    }
}

public static Shape getShapeFromRectangle(Rectangle rectangle){
    PolygonShape polygonShape = new PolygonShape();
    polygonShape.setAsBox( hx: rectangle.width*0.5F/ TILE_SIZE, hy: rectangle.height*0.5F/ TILE_SIZE);
    return polygonShape;
}
public static Vector2 getTransformedCenterForRectangle(Rectangle rectangle){
    Vector2 center = new Vector2();
    rectangle.getCenter(center);
    return center.scl(1/TILE_SIZE);
}
```

Obr. 3.1

6. Tiled

Pro stvoření herního světa jsme použili program Tiled. Využívá systém dlaždic (zmíněn v sekci 9. Grafická stránka projektu). Pomocí Tiledu jsme vytvořili vzhled i terén světa. Terénem je myšlen Hitbox. Hitbox zajišťuje při správném použití systém kolizí. Jedná se o hranici, po kterou sahá velikost daného objektu. Pokud postava hráče, projektil nebo jiný objekt narazí svým Hitboxem na Hitbox jiného objektu, se kterým má možnost kolidovat, narazí do něj a nebude moci dále pokračovat v cestě. Celý systém Hitboxů zajišťuje, že hráč musí projít bludištěm a nemůže jít, jak se mu zachce přes stěny. V našem případě byly Hitboxy tvořeny pomocí objektů typu *Rectangle*. Jedná se o objekt z programu Tiled. Má vlastnosti buď bodu, přímky, nebo pravidelného čtyřúhelníku. Tento objekt je ve finálním vykreslení hry neviditelný. Z těchto "rectanglů" se následně skládá samotný Hitbox mapy (viz obr. 3.2).



Obr. 3.2

7. Ničitelné objekty

Tyto objekty jsou tvořeny také pomocí metody *public void builBuildingsBodies()*, ale je tu důležité nastavit *userData*. Pomocí *userData* můžeme rozlišovat, co do sebe naráží a v případě kontaktu určitá těla "zničit". Když chceme "zničit" objekt, potřebujeme se zbavit "fixture" a s tím spojené i "body". Na zničení "fixture" můžeme zavolat metodu *destroyFixture()* a na odstraňování těla metodu *destroyBody()*. Libgdx ale nedovoluje níčení "fixture" ani "body" za průběhu aplikace, když se volá metoda *step()* nebo metoda *render()*. Proto jsme museli objekty do "seznamu objektů k zničení", který pak projde metoda, která se volá po metodě *step()* a před metodou *render()*.

```
public static Array<Fixture> fixturesToRemove = new Array<>();
public static Array<Body> bodiesToRemove = new Array<>();

private void removeFromWorld(Fixture fixture){
    fixturesToRemove.add(fixture);
    if(fixture.getBody().getFixtureList().size - 1 <1){
        bodiesToRemove.add(fixture.getBody());
    }
}

public void removeObjects (World world) {
    for (Fixture fixture : fixturesToRemove) {
        fixture.getBody().destroyFixture(fixture);
        fixturesToRemove.removeValue(fixture, identity: true);
    }
    for (Body body : bodiesToRemove) {
        world.destroyBody(body);
        bodiesToRemove.removeValue(body, identity: true);
    }
}
```

8. Sběratelné objekty

Sběratelné objekty fungují úplně stejně jako ničitelné objekty až na to, že když se zničí, nastaví boolean hodnotu *isPickupable* na *true* a tím tak sdělí invetáři, aby se tam objevil.

9. Inventář

K vytvoření inventáře jsme použili *ImageButtons*. Přes 2 for-cykly jsme vytvořili čtverec *ImageButtonů*, pro které jsme nastavili *EventListenery*. Do 11 buttonů z 16 jsme načetli texturu předmětu, který se dá sbírat a nastavili jsme, aby nebyly vidět. Poté, jakmile se sběratelné předměty zničily a *isPickupable* mělo hodnotu *true*, byly předměty zobrazeny v inventáři.

10. Uživatelský vstup

Uživatelský vstup se dal řešit dvěma způsoby:

1. podmínky se vloží do metody *render()*, která by řešila vstup z klávesnice (viz obr. 4)
2. námi zvolené vytvoření třídy *KeyBoardController*, která pak implementuje interface *InputProcessor* (viz obr. 5)

```
@Override
public void render() {
    Gdx.gl.glClearColor(1, 1, 1, 1);
    Gdx.gl.glClear(GL10.GL_COLOR_BUFFER_BIT);

    if(Gdx.input.isKeyPressed(Input.Keys.LEFT)){
        if(Gdx.input.isKeyPressed(Input.Keys.CONTROL_LEFT))
            sprite.translateX(-1f);
        else
            sprite.translateX(-10.0f);
    }
    if(Gdx.input.isKeyPressed(Input.Keys.RIGHT)){
        if(Gdx.input.isKeyPressed(Input.Keys.CONTROL_LEFT))
            sprite.translateX(1f);
        else
            sprite.translateX(10.0f);
    }
    batch.begin();
    sprite.draw(batch);
    batch.end();
}
```

Obr. 4

```
public class MyInputProcessor implements InputProcessor {
   public boolean keyDown (int keycode) {
      return false;
   }

   public boolean keyUp (int keycode) {
      return false;
   }

   public boolean keyTyped (char character) {
      return false;
   }

   public boolean touchDown (int x, int y, int pointer, int button) {
      return false;
   }

   public boolean touchUp (int x, int y, int pointer, int button) {
      return false;
   }

   public boolean touchDragged (int x, int y, int pointer) {
      return false;
   }

   public boolean mouseMoved (int x, int y) {
      return false;
   }
```

Obr. 5

Instance třídy jménem *controller* se pak nachází ve třídě *B2dModel*, která uživatelský vstup řeší v metodě *logicStep()* (viz obr. 6).

```
public void logicStep(float delta){
    if(controller.isMouse1Down && pointIntersectsBody(player,controller.mouseLocation)){
        System.out.println("Player was clicked");
    }
    if(controller.isMouse1Down){
        sword.setTransform(sword.getPosition(), angle: -90*BodyFactory.DEGTORAD);
        swordIsUp = false;
    }
    else if (!swordIsUp){
        sword.setTransform(sword.getPosition(), angle: 0);
    }

    if(controller.left){
       player.applyForceToCenter( forceX: -10, forceY: 0, wake: true);

    }else if(controller.right){
        player.applyForceToCenter( forceX: 10, forceY: 0, wake: true);

    }else if(controller.up){
        player.applyForceToCenter( forceX: 0, forceY: 10, wake: true);

    }else if(controller.down){
        player.applyForceToCenter( forceX: 0, forceY: -10, wake: true);

    }else{
        player.setLinearVelocity( vX: 0, vY: 0);
    }

    if(Gdx.input.justTouched()){
        listOfProjectiles.add(new ProjectileEntity( x: player.getPosition().x +2, y: player.getPosition().y+2));
    }

    world.step(delta , velocityIterations: 3, positionIterations: 3);

}
```

Obr. 6

Hráč se může pomocí standardních klávesnic W, A, S, D pohybovat nahoru, doleva, dolu, doprava. Taky pomocí klávesnice E může hráč otevřít inventář.

```
private void changeToInvetoryScreen(){
    if(Gdx.input.isKeyJustPressed(Input.Keys.E)){
        parent.changeScreen(MyGdxGame.INVENTORY);
    }

}
```

11. Správce souborů

Framework LibGdx obsahuje třídu AssetManager, která umožňuje načíst soubory před načtením hry. V budoucnu lze takto načíst jednotlivé části hry místo celé hry.

Třídu *B2dAssetManager* (viz obr. 7) tvoří metody, které načítají různé soubory, jako jsou textury postav, zbraní, popřípadě zvuky a hudbu. Důležité je, aby všechny tyto soubory byly ve složce *„Assets“* uvnitř složky *„Android“*, jinak se občas stane, že něco nefunguje (např. načítání mapy).

Zvažovali jsme použití TexturePackeru, který sjednotí více textur do jednoho souboru, typu *atlas,* a pak v tom souboru načte jistý region. Pro zjednodušení jsme se však rozhodli tento krok vynechat.

```
public class B2dAssetManager {

    public final AssetManager manager = new AssetManager();

    public final String playerImage = "images/player.png";
    public final String enemyImage = "images/enemy.png";
    public final String boingSound = "sounds/boing.wav";
    public final String pingSound = "sounds/ping.wav";
    public final String sword = "images/GoldenSpear.png";
    public final String skin = "skin/glassy-ui.json";
    public final String playingSong = "music/Vitas_7th_Element.mp3";
    public final String background = "skin/EternalBackground.png";
    public final String menuBackGround = "skin/menuBackGround.png";
    public final String fireball = "images/fireball.png";
    public final String ugandaBoi = "images/ugandaBoi.png";
    public final String Jellik = "images/Jellik.png";
    public final String Inventory = "skin/Inventory.png";
    public final String boomSound = "sounds/boom.wav";
    public final String health = "images/blank.png";

    public void queueAddMusic(){
        manager.load(playingSong, Music.class);
        manager.load(boomSound,Music.class);
    }
    public void queueAddImages(){
        manager.load(playerImage, Texture.class);
        manager.load(enemyImage, Texture.cl
        manager.load(sword, Texture.class);
        manager.load(fireball,Texture.class);
        manager.load(ugandaBoi,Texture.class);
```

Obr. 7

Obr. 7

12. Obrazovky

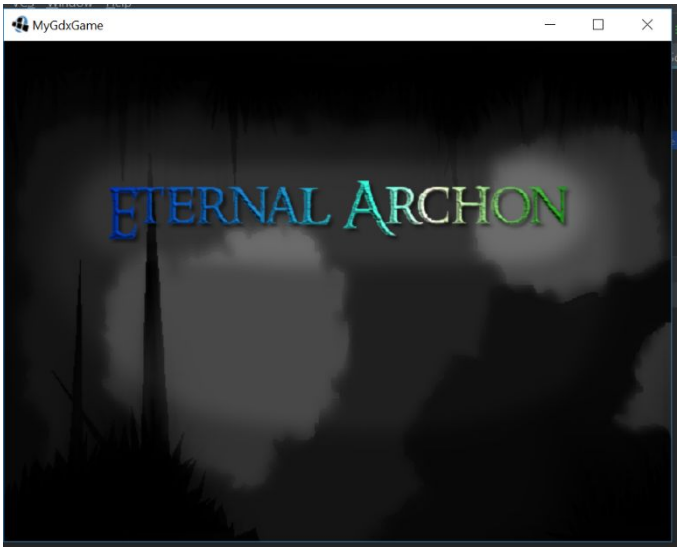
Aby bylo možné přepínání mezi obrazovkami, vytvořili jsme třídu *MyGdxGame* jako potomka třídy *Game,* a pro třídy *LoadingScreen, MainScreen, MenuScreen* a *PreferencesScreen* jsme naimplementovali interface *Screen*. Metoda *changeScreen()* pak zajišťuje přepínání (viz obr. 8).

```
public void changeScreen(int screen) {
    switch (screen) {
        case MENU:
            if (menuScreen == null) menuScreen = new MenuScreen( main: this);
            this.setScreen(menuScreen);
            break;
        case PREFERENCES:
            if (preferencesScreen == null) preferencesScreen = new PreferencesScreen( main: this);
            this.setScreen(preferencesScreen);
            break;
        case APPLICATION:
            if (mainScreen == null) mainScreen = new MainScreen(this);
            this.setScreen(mainScreen);
            break;
        case INVENTORY:
            if(inventoryScreen == null) inventoryScreen = new InventoryScreen( main: this);
            this.setScreen(inventoryScreen);
            break;
    }
}
```

Obr. 8

12.1. LoadingScreen

Načítací obrazovka (viz obr. 9) načítá soubory ke hře. Jelikož jsme již vytvořili správce souborů, stačí zavolat jeho metody a naše soubory se načtou před začátkem hry. Pomocí třídy *Spritebatch* se pak vykreslí pozadí načítací obrazovky.



Obr. 9

12.2. MenuScreen

Poté, co se načtou soubory, zobrazí se menu s třemi tlačítky: *„Play“*, *„Preferenc“* a *„Exit“* (viz obr. 10). Vytvoření těchto tří tlačítek zajišťuje instance třídy *Stage* a instance *Table*.

Metoda *addActor()* přidává již zmíněnou instanci třídy *Table*. V ní jsou další 3 instance třídy *TextButton*. Pro ovládání hry klávesnicí a myší jsme přidali každému tlačítku *eventListener*. Aby se pak vše zobrazilo, volá metoda *render()* metodu *stage.draw()* (viz obr. 11).

```
@Override
public void show() {
    Gdx.input.setInputProcessor(stage);
    Table table = new Table();
    table.setFillParent(true);
        table.setDebug(true);
        stage.addActor(table);


    TextButton newGame = new TextButton( text: "New Game", skin);
    TextButton preferences = new TextButton( text: "Preferences", skin);
    TextButton exit = new TextButton( text: "Exit", skin);

    table.add(newGame).fillX().uniformX();
    table.row().pad( top: 10, left: 0, bottom: 10, right: 0);
    table.add(preferences).fillX().uniformX();
    table.row();
    table.add(exit).fillX().uniformX();

    exit.addListener((ChangeListener) (event, actor) -> { Gdx.app.exit(); });

    newGame.addListener((ChangeListener) (event, actor) -> {
            parent.changeScreen(MyGdxGame.APPLICATION);
    });

    preferences.addListener((ChangeListener) (event, actor) -> {
            parent.changeScreen(MyGdxGame.PREFERENCES);
    });

}
```

Obr. 10

```
@Override
public void render(float delta) {
    Gdx.gl.glClearColor( red: 0f, green: 0f, blue: 0f, alpha: 1);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

    stage.act(Math.min(Gdx.graphics.getDeltaTime(), 1 / 30f));
    stage.getBatch().begin();
    stage.getBatch().draw(backgroundTexture, x: 0, y: 0,Gdx.graphics.getWidth(), Gdx.graphics.getHeight());
    stage.getBatch().end();
    stage.draw();
}
```

Obr. 11

12.3. InventoryScreen

V této třídě se vykresluje inventář hráče. Důvod, proč jsme to neudělali jako stage ve třídě MainScreen, je jednoduchost. Takhle nemusíme řešit střídání inputu a outputu, taky když z toho takhle uděláme samostatnou obrazovku tak se zastaví celá hra.

12.4. MainScreen

V hlavní obrazovce se odehrává nejdůležitější část - a to samotná hra. Vše se zde vykresluje.

Jednou z vykreslovaných věcí je mapa hry. Pro načtení mapy je použita instance třídy *OrthographicCamera*, jelikož pohled z této kamery je ideální pro 2D hry. Dále jsme potřebovali samotný objekt mapy, třídu *TiledMap*, a pro vykreslení mapy třídu *TiledMapRenderer*.

Vše vypadalo v pořádku, dokud se nezačaly objevovat 2 chyby. Jedním z problémů byla serializační chyba, což byla chyba se špatným formátem souboru. Druhým problémem bylo, že projekt nedokázal naši mapu načíst. Řešením pro oba problémy bylo umístit mapu do správného adresáře.

Poté začal program vypisovat jiný error. Tento error hlásil, že chybí jakési dokumenty. Uvědomili jsme si, že do adresáře asset patří všechno, co se týkalo té mapy (tzn. i všechny dokumenty typu *.tsx* a *.png*). Po implementaci se pak mapa konečně načetla.

Pro vykreslení ostatních objektů jsme opět použili třídu *SpriteBatch* a metodu *draw()*.

12.4.1. Projectile

Jednou z mechanik hry je možnost hráče střílet tam, kam klikne myš. To zajišťuje třída *Projectile*. Tento problém se dal řešit třemi způsoby.

Nejdříve jsme vyřešili otázku vertikálního letu střely. Aby letěla vertikálně, stačilo k hodnotě *y* přičítat chtěnou vzdálenost. Pro vykreslení projektilů se ve třídě *MainScreen* vytvořil *ArrayList*, do kterého jsme přidávali naše projektily. Všechny projektily, co byly v ArrayListu se pak vykreslily.

Samozřejmě jsme chtěli, aby naše projektily po určité vzdálenosti zmizely. Pro tento účel jsme vytvořili další *ArrayList*, protože nám problém hlásil, že nemůže odebírat a přidávat naše projektily zároveň.

Poté, co jsme vyřešili vertikální pohyb, začali jsme zkoumat způsoby pro klik na myši. Našli jsme 2 matematické způsoby:

- přes výpočet úhlu vůči hráči a cíli (pomocí arkus tangens)
- Vektory

Po vyzkoušení všech metod fungovaly vektory nejlépe a bylo méně řádků kódu. Ani tak to nebylo ideální kvůli odlišným souřadnicovým systémům myši a zobrazování textur. Souřadnicový systém myši má počátek vlevo nahoře, kdežto zobrazování textur má počátek uprostřed okna. Tento problém jsme vyřešili tak, že jsme jednu soustavu změnili, aby se rovnala té druhé, pomocí těchto řádků kódu:

```
screenCoordinates.set(Gdx.input.getX(),Gdx.input.getY(), z: 0);
worldCoordinates.set(screenCoordinates);
cam.unproject(worldCoordinates);
```

13. Grafická stránka projektu:

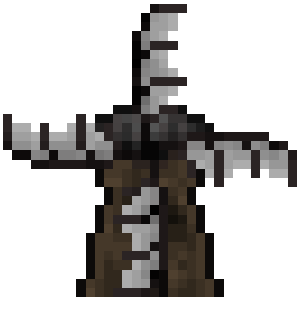
Vzhled našeho projektu se skládá z jednotlivých obrázků tvořených převážně ve webové aplikaci *Piskel*. *Piskel* je open-source editor pro pixel art. Umožňuje tvorbu jednoduchých obrázků, animací a jejich import a export v mnohých formátech. Aplikace má jednoduché uživatelské rozhraní a mnohé užitečné klávesové zkratky - její ovládání je tedy velice jednoduché a intuitivní.

13.1. Tvorba textur

Tvorba jedné textury trvá 15 minut až několik hodin. Čas se odvíjí od velikosti a typu. Například zbraň je na tvorbu výrazně snazší, než budova nebo character (viz obr. 13-15).

Časově náročnější jsou pak *animace*. Důvodem je to, že každá animace je tvořena z jednotlivých obrázků, které musí autor postupně vytvořit. Proto tvorba složitějších animací, jako například vlaku (viz projekt samotný), trvá i několik hodin.

Další složitou záležitostí při tvorbě vzhledu hry je např. *textura země*. Pokud chce autor mít „na pohled hezky vypadající výsledek“, musí svůj výtvor optimalizovat pro spojení dohromady z toho důvodu, že jednotlivě vykreslovat každý pixel velkého herního plánu by byl úkol skoro lidsky nemožný. Proto se používá *technika dlaždic.* Autor vytvoří texturu pro jednu, či více dlaždic, ze kterých následně poskládá celou mapu. Výhodou tohoto způsobu je následné usnadnění programování projektu. Hra totiž potom může fungovat trochu více jako např. šachy. Další výhodou tohoto stylu je možné využití nástroje, jako např. *Tiled*. Tento nástroj umožňuje tvorbu map, jejich vrstev, objektů a jednoduchého vložení do projektu.



Obr. 14

Obr. 13

Obr. 15

14. ZÁVĚR

Díky této ročníkové práci jsme si procvičili práci v Javě a naučili se spoustu nových věcí o tom, jak se tvoří hry. Také jsme se naučili pracovat s nástroji pro grafiku a tvorbu textur.

K této hře bychom se v budoucnu chtěli ještě vrátit, dodělat věci, které jsme nestihli a určitě ji ještě vylepšit.

15. ZDROJE:

1. *libGDX. Dostupné z URL:* <https://libgdx.badlogicgames.com/>
2. *Objects/Bodies.* Dostupné z URL: <https://github.com/libgdx/libgdx/wiki/box2d#objectsbodies/>
3. *Full LibGDX Game Tutorial – Box2D Body Factory* Dostupné z URL: <https://www.gamedevelopment.blog/full-libgdx-game-tutorial-box2d-body-factory/>
4. *Event driven keyboard and mouse handling.* Dostupné z URL: <https://www.gamefromscratch.com/post/2013/10/15/LibGDX-Tutorial-4-Handling-the-mouse-and-keyboard.aspx/>
5. *Full LibGDX Game Tutorial – Input Controller.* Dostupné z URL: <https://www.gamedevelopment.blog/full-libgdx-game-tutorial-input-controller/>
6. *Full LibGDX Game Tutorial – Rendering and Asset manager.* Dostupné z URL: <https://www.gamedevelopment.blog/full-libgdx-game-tutorial-rendering-asset-manager/>
7. *Full LibGDX Game Tutorial – Loading Screen.* Dostupné z URL: <https://www.gamedevelopment.blog/full-libgdx-game-tutorial-loading-screen/>
8. *Firing bullet from gun's current position to mouse x & y position.* Dostupné z URL: <http://www.java-gaming.org/topics/firing-bullet-from-gun-s-current-position-to-mouse-x-y-position/35020/view.html>
   Shooting bullet towards mouse position in Java/Slick2D. Dostupné z URL: <https://stackoverflow.com/questions/34122380/shooting-bullet-towards-mouse-position-in-java-slick2d>
9. Brent Aureli. *Demo Multiplayer LibGDX game using a NodeJS server and Socket.IO for event firing*. Dostupné z URL: <https://github.com/BrentAureli/MultiplayerDemo>
10. *Socket.IO-client Java.* Dostupné z URL: <https://github.com/socketio/socket.io-client-java>
11. *Create Box2D Bodies from Tiled Map.* Dostupné z URL: <https://riptutorial.com/libgdx/example/17831/create-box2d-bodies-from-tiled-map?fbclid=IwAR2AspPviiOLyooGTSkEOY-xAElSMZSTYS5UAqWs9gC0FG0EQkGaz-KEbwU>
12. *Breakable objects in Box2D.* Dostupné z URL: <https://www.youtube.com/watch?v=GM6s8v7_oA8&fbclid=IwAR0h28EC9KQis-3A9cFsK_JLsZekXBHzHxooSA1gwYcu1tWQIYzA3N-fDcs>
13. *Turn off collisions one body.* Dostupné z URL: <https://badlogicgames.com/forum/viewtopic.php?f=11&t=1352&fbclid=IwAR02NrcmFot5AOXL6jGN-JnlTH8MddNCJAxTi7gc1QF-nYZLWWNjTuu6F7s>
14. *LibGDX autoresizing image button.* Dostupné z URL: <https://stackoverflow.com/questions/22788726/libgdx-autoresizing-image-button>

OBRÁZKY:

- *Obr. 6*. Dostupný z URL: <https://github.com/libgdx/libgdx/wiki/Event-handling>