|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Znalezione obrazy dla zapytania utp logo | Uniwersytet Technologiczno-Przyrodniczy im. J. J. Śniadeckich w Bydgoszczy  Wydział Telekomunikacji, Informatyki i Elektrotechniki Zakład Systemów Teleinformatycznych | | | Znalezione obrazy dla zapytania wtiie logo | |
| **Przedmiot** | *Programowanie aplikacji mobilnych* | | | **Kierunek/ Tryb** | IS/ST |
| **Grupa** | 2 | **Data oddania** | 21-05-2019 | **Imię Nazwisko** | Dominik Kowalski  Piotr Januszewski |
| **Opis Projektu** | Gra Super Mario Bros na urządzenia z systemem Android | | |

1. Ogólny opis gry – koncepcja

Celem projektu było napisanie gry wzorując się na istniejącej już grze Super Mario Bros. Gra miała być przeznaczona na urządzenia z systemem Android.

Charakterystyka gry:

* Gra jednoosobowa
* Występujące postaci oraz przedmioty:

Postać główna, którą steruje użytkownik

* mario

Wrogowie:

* goomba
* żółw

Przedmioty:

* grzyb

Elementy świata:

* bloczki z monetami
* bloczki z grzybami
* bloczki z cegły
* rury
* zamek kończący etap

Granice systemu gry:

Obszarem gry jest mapa o szerokości 240 jednostek oraz wysokości 13 jednostek , gdzie wartość jednostki odpowiada 16px. Użytkownik steruje postacią za pomocą trzech przycisków (ruch w lewo, ruch w prawo oraz skok). Poniżej umieszczone są grafiki reprezentujące owe przyciski.



Lista możliwości (Funkcji gry)

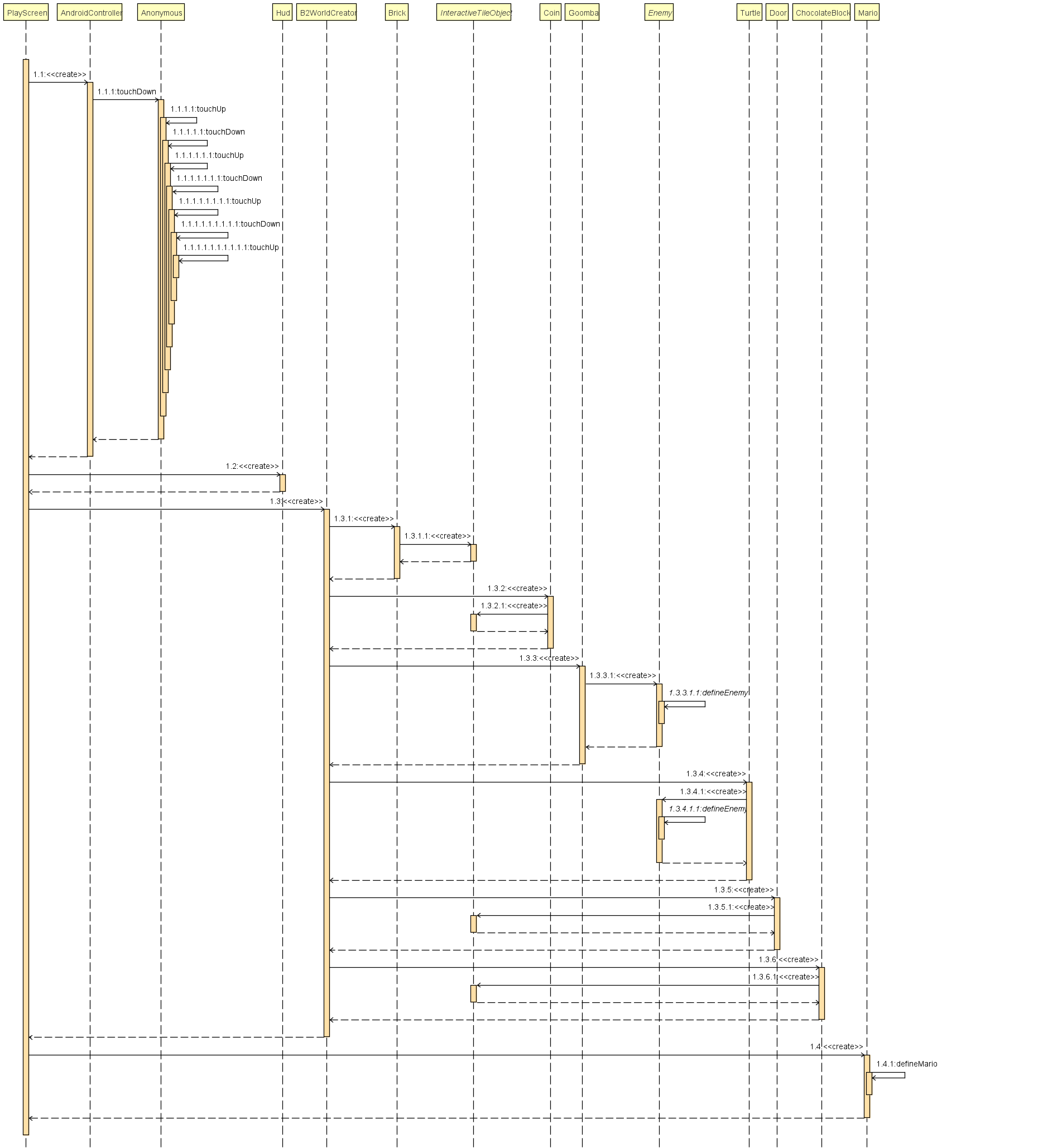
1. Zdobycie grzyba wchodząc w niego
2. Zdobycie monety uderzając głową w blok z monetą
3. Utrata życia spowodowana kolizją z jednym z dwóch przeciwników
4. Utrata życia spowodowana wpadnięciem w przepaść
5. Likwidacja przeciwnika (goomba) poprzez naskoczenie na niego
6. Przemienienie przeciwnika (żółwia) w stan skorupy na 5 sekund poprzez naskoczenie na niego
7. Wprawienie żółwia w postaci skorupy w ruch poprzez naskoczenie na niego
8. Zatrzymanie żółwia w postaci skorupy poprzez naskoczenie na niego
9. Likwidacja wrogów przez skorupę żółwia wprawioną w ruch
10. Ukończenie mapy poprzez wejście do zamku
11. Analiza dziedziny

Diagram klas wraz z relacjami:

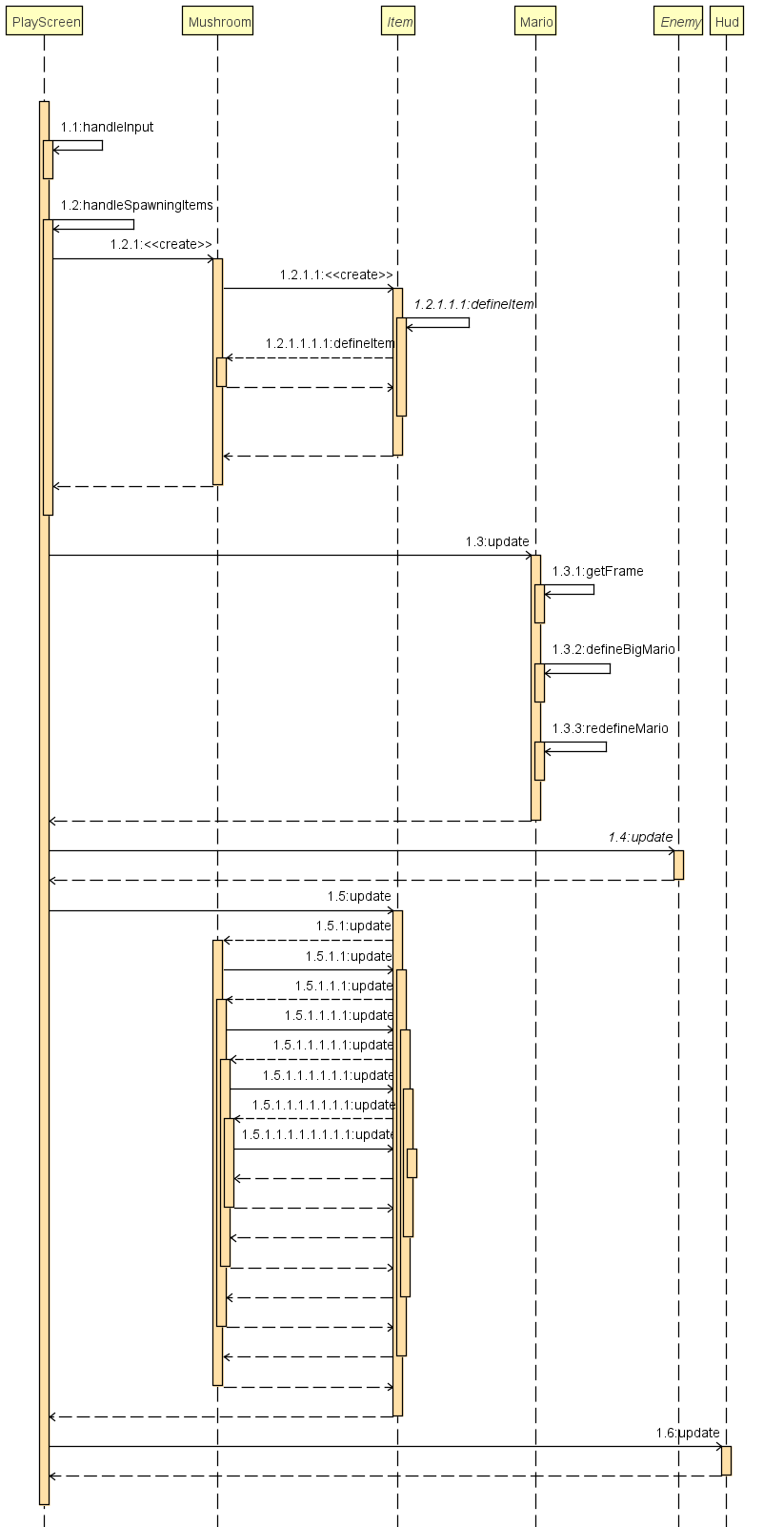


Diagramy sekwencji:

Konstruktor klasy PlayScreen:



Metoda update klasy PlayScreen:

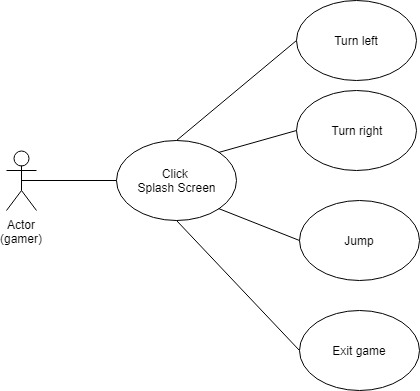


1. Specyfikacja wymagań

Przypadki użycia:

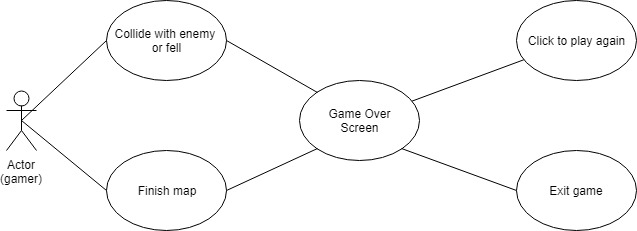
Możliwości w grze:

Warunki: Użytkownik włącza grę



Scenariusze w grze:

Warunki: Użytkownik jest już w trakcie rozgrywki



IV. Analiza i projekt

1. Architektura systemu gry

Warstwy:

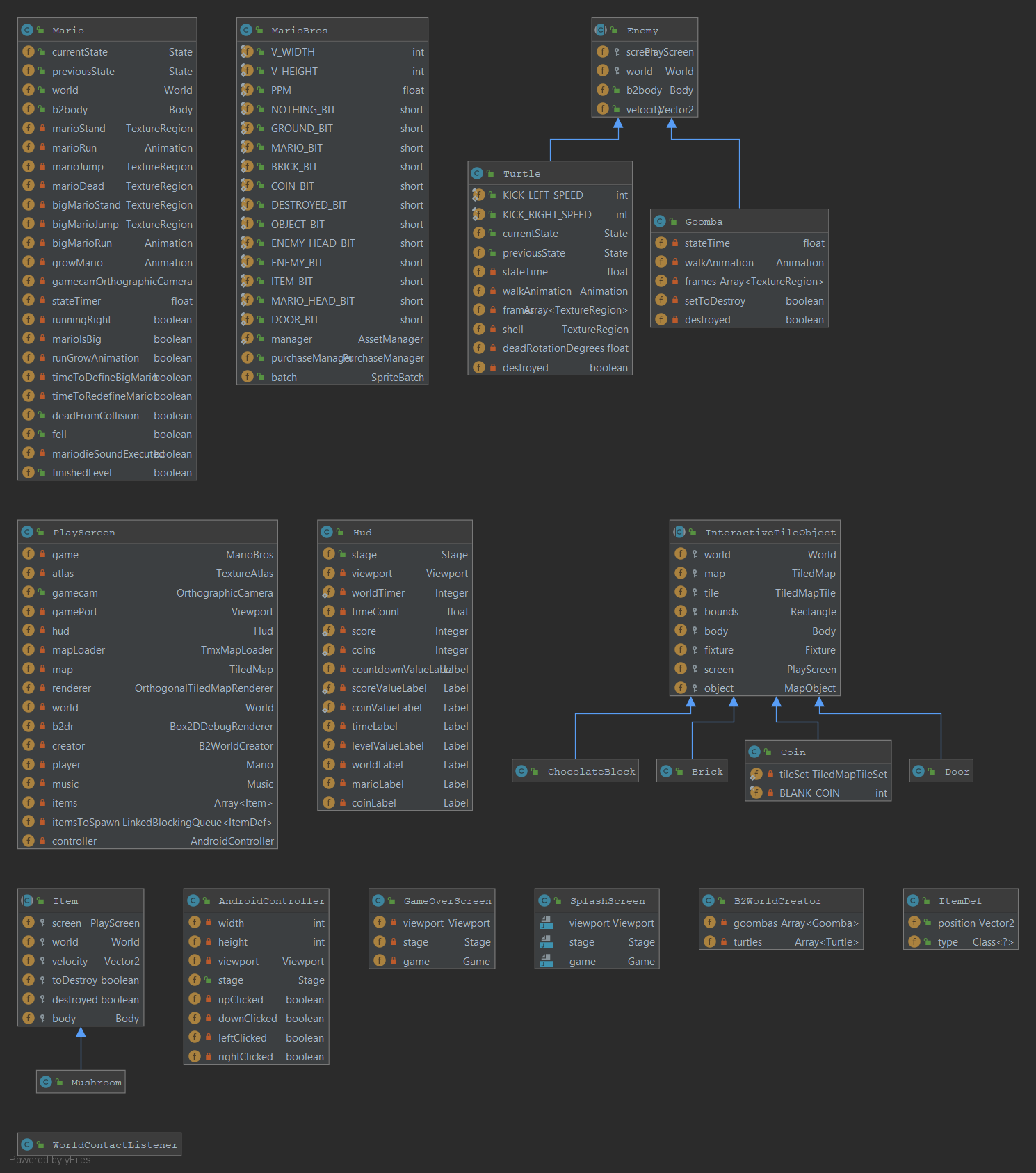
* Widok
* Logika

Warstwy odpowiedzialne za widok znajdują się w pakietach: Screens oraz Scenes.

Warstwy odpowiedzialne za logikę znajdują się w pozostałych pakietach.

2. Obiektowy model analizy

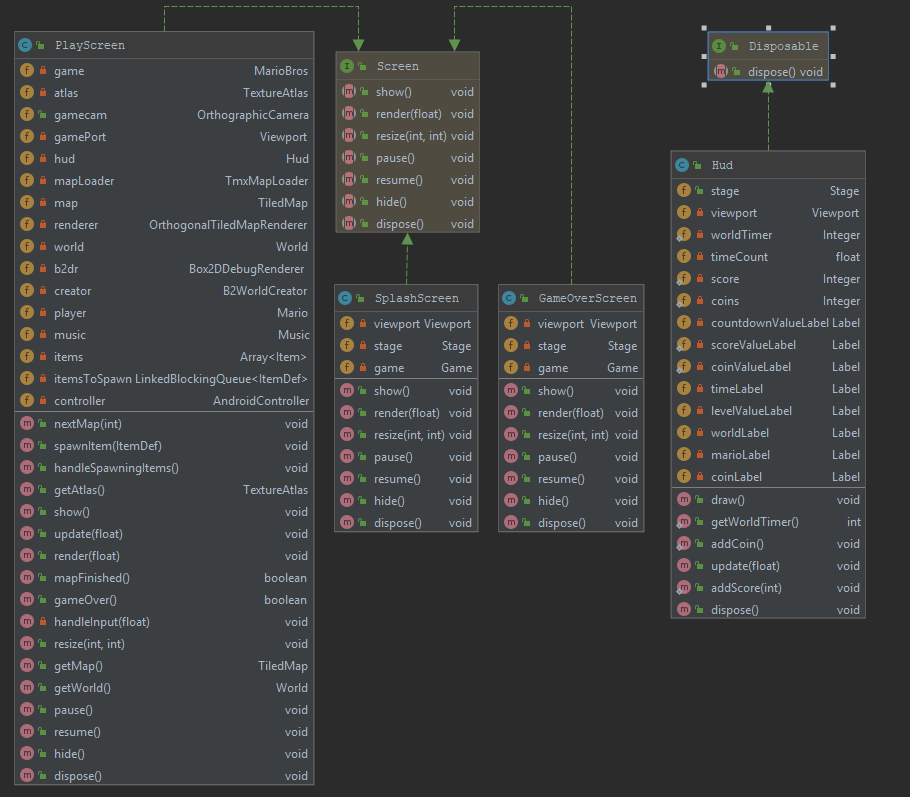
Diagramy klas wraz z atrybutami:



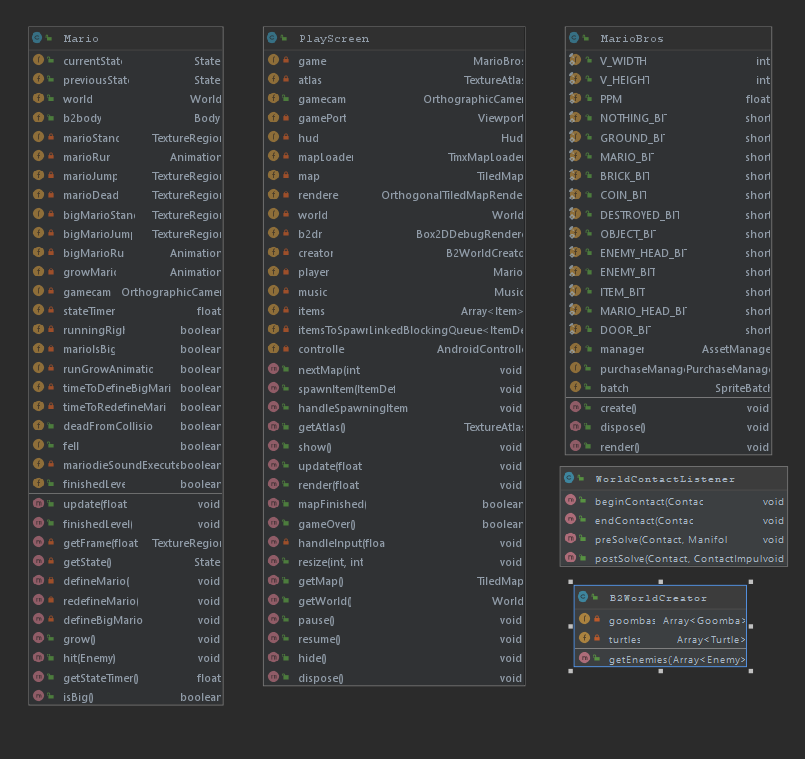
1. Projekt oprogramowania

Projekt ten został zrealizowany z przeznaczeniem na platformę Android. Język jakiego użyto to Java.

(1-2) Klasy odpowiedzialne za prezentacje danych wraz atrybutami, metodami oraz relacjami pomiędzy nimi:

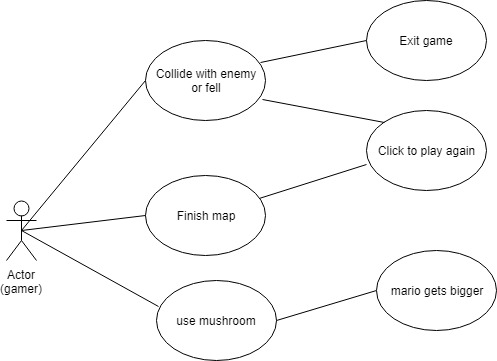


Klasy odpowiedzialne za logikę gry wraz atrybutami oraz metodami:



3) Przypadki użycia odpowiadającym scenariuszom

Gdy Mario jest mały:



Gdy Mario jest duży:

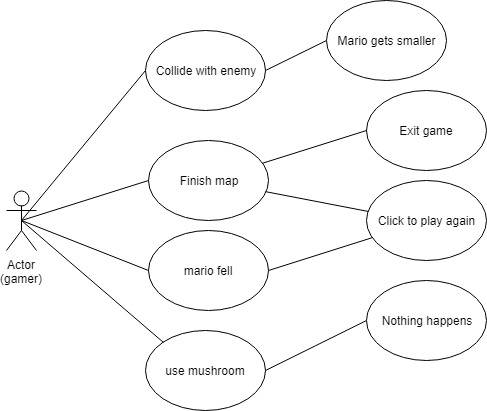
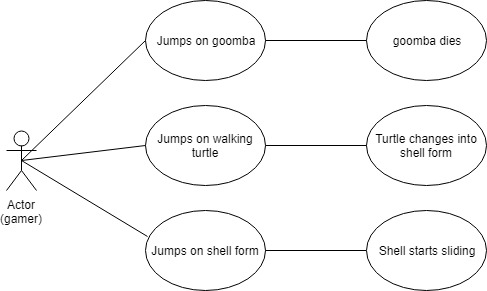


Diagram w przypadku gdzie wielkość Mario nie ma znaczenia:



Projekt interfejsu użytkownika IRS



Użytkownik ma do dyspozycji 3 przyciski w grze. Strzałkę w lewo odpowiedzialną za ruch mario w lewo, strzałkę w prawo odpowiedzialną za ruch mario w prawo oraz strzałkę w górę odpowiedzialną za skok mario.

Przyciski zostały umieszczne w optymalny sposób tj. tak aby nie kolidowały one z klawiszami funkcyjnymi urządzeń mobilnych.



V. Implementacja

Wybrane fragmenty kodu wraz z opisem najważniejszych funkcjonalności:

Poniżej znajduje się ciało klasy MarioBros, która jest główną klasą programu. W metodzie create

Obiekty zostają zainicjalizowane oraz przypisane do zmiennych klasowych. Po załadowaniu plików dźwiękowych to obiektu klasy AssetManager wykonywana zostaje metoda setScreen, gdzie jako argument przekazany zostaje ekran, który ma zostać wyświetlony.

public static AssetManager *manager*;  
public PurchaseManager purchaseManager;  
public SpriteBatch batch;  
  
@Override  
public void create() {  
 batch = new SpriteBatch();  
 *manager* = new AssetManager();  
 *manager*.load("audio/music/mario\_music.ogg", Music.class);  
 *manager*.load("audio/sounds/coin.wav", Sound.class);  
 *manager*.load("audio/sounds/bump.wav", Sound.class);  
 *manager*.load("audio/sounds/breakblock.wav", Sound.class);  
 *manager*.load("audio/sounds/powerup\_spawn.wav", Sound.class);  
 *manager*.load("audio/sounds/powerup.wav", Sound.class);  
 *manager*.load("audio/sounds/powerdown.wav", Sound.class);  
 *manager*.load("audio/sounds/stomp.wav", Sound.class);  
 *manager*.load("audio/sounds/mariodie.wav", Sound.class);  
 *manager*.load("audio/sounds/level\_complete.mp3", Sound.class);  
 *manager*.finishLoading();  
 setScreen(new SplashScreen(this));  
}

Klasa B2WorldCreator. W niej tworzony zostaje świat oraz tworzone obiekty na podstawie mapy stworzonej w programie Tiled. Każdy obiekt na mapie jest przypisany do odpowiedniej warstwy po czym zostaje on mapowany do obiektów Javy w sposób jaki został ukazany w poniższym kodzie.

public class B2WorldCreator {  
  
 private Array<Goomba> goombas;  
 private Array<Turtle> turtles;  
  
  
 public B2WorldCreator(PlayScreen screen) {  
 World world = screen.getWorld();  
 Map map = screen.getMap();  
 BodyDef bdef = new BodyDef();  
 PolygonShape shape = new PolygonShape();  
 FixtureDef fdef = new FixtureDef();  
 Body body;  
  
 //ground  
 for (MapObject object : map.getLayers().get(5).getObjects().getByType(RectangleMapObject.class)) {  
 Rectangle rect = ((RectangleMapObject) object).getRectangle();  
 bdef.type = BodyDef.BodyType.*StaticBody*;  
 bdef.position.set((rect.getX() + rect.getWidth() / 2) / MarioBros.*PPM*, (rect.getY() + rect.getHeight() / 2) / MarioBros.*PPM*);  
 body = world.createBody(bdef);  
 shape.setAsBox(rect.getWidth() / 2 / MarioBros.*PPM*, rect.getHeight() / 2 / MarioBros.*PPM*);  
 fdef.shape = shape;  
 fdef.filter.categoryBits = MarioBros.*GROUND\_BIT*;  
 body.createFixture(fdef);  
 }

//brick  
 for (MapObject object : map.getLayers().get(3).getObjects().getByType(RectangleMapObject.class)) {  
 new Brick(screen, object);  
 }

Klasa Odpowiedzialna za kolizje w grze.

Każdy obiekt w grze, który ma kolidować z innymi obiektami ma swoją własną reprezentację w postaci zmiennej int. Za pomocą tych zmiennych określane zostaje również z jakimi obiektami ma kolidować dany obiekt. Kolizje sprawdzane są w poniższej klasie.

public class WorldContactListener implements ContactListener {  
  
 @Override  
 public void beginContact(Contact contact) {  
 Fixture fixA = contact.getFixtureA();  
 Fixture fixB = contact.getFixtureB();  
 int cDef = fixA.getFilterData().categoryBits | fixB.getFilterData().categoryBits;  
  
  
 switch (cDef) {  
 case MarioBros.*MARIO\_HEAD\_BIT* | MarioBros.*BRICK\_BIT*:  
 case MarioBros.*MARIO\_HEAD\_BIT* | MarioBros.*COIN\_BIT*:  
 if (fixA.getFilterData().categoryBits == MarioBros.*MARIO\_HEAD\_BIT*) {  
 ((InteractiveTileObject) fixB.getUserData()).onHeadHit((Mario) fixA.getUserData());  
 } else {  
 ((InteractiveTileObject) fixA.getUserData()).onHeadHit((Mario) fixB.getUserData());  
 }  
 break;  
 case MarioBros.*ENEMY\_HEAD\_BIT* | MarioBros.*MARIO\_BIT*:  
 if (fixA.getFilterData().categoryBits == MarioBros.*ENEMY\_HEAD\_BIT*) {  
 ((Enemy) fixA.getUserData()).hitOnHead((Mario) fixB.getUserData());  
 } else {  
 ((Enemy) fixB.getUserData()).hitOnHead((Mario) fixA.getUserData());  
 }  
 Gdx.*app*.log("Collision", "enemy head");  
 break;

...

}

Przykładowe określenie reprezentacji bitowej obiektu oraz tego z jakimi obiektami ma on kolidować.

Zostaje to zrealizowane za pomocą przypisania odpowiedniej wartości do zmiennej filter oraz maskBits określa z jakimi grupami obiektów ma kolidować dany obiekt.

public class Mushroom extends Item {  
  
 @Override  
 public void defineItem() {  
 BodyDef bdef = new BodyDef();  
 bdef.position.set(getX(), getY());  
 bdef.type = BodyDef.BodyType.*DynamicBody*;  
 body = world.createBody(bdef);  
  
 FixtureDef fdef = new FixtureDef();  
 CircleShape shape = new CircleShape();  
 shape.setRadius(6 / MarioBros.*PPM*);  
 fdef.filter.categoryBits = MarioBros.*ITEM\_BIT*;  
 fdef.filter.maskBits = MarioBros.*MARIO\_BIT* |  
 MarioBros.*OBJECT\_BIT* |  
 MarioBros.*GROUND\_BIT* |  
 MarioBros.*COIN\_BIT* |  
 MarioBros.*BRICK\_BIT*;  
  
 fdef.shape = shape;  
 body.createFixture(fdef).setUserData(this);  
 }

Metoda update klasy Mario, która odpowiedzialna jest za całą mechanikę postaci głównej.

public void update(float dt) {  
 if (b2body.getPosition().y < gamecam.position.y - gamecam.viewportHeight / 2) {  
 if (!mariodieSoundExecuted && !deadFromCollision) {  
 MarioBros.*manager*.get("audio/music/mario\_music.ogg", Music.class).stop();  
 MarioBros.*manager*.get("audio/sounds/mariodie.wav", Sound.class).play();  
 mariodieSoundExecuted = true;  
 }  
 fell = true;  
 }  
if ((b2body.getPosition().x - getWidth() / 2 <= gamecam.position.x - gamecam.viewportWidth / 2)) {  
 b2body.setTransform(new Vector2(gamecam.position.x - (gamecam.viewportWidth / 2) + getWidth() / 2, b2body.getPosition().y), 0);  
 b2body.setLinearVelocity(new Vector2(0, b2body.getLinearVelocity().y));  
 }  
 if (marioIsBig) {  
 setPosition(b2body.getPosition().x - getWidth() / 2, b2body.getPosition().y - getHeight() / 2 - 6 / MarioBros.*PPM*);  
 } else {  
 setPosition(b2body.getPosition().x - getWidth() / 2, b2body.getPosition().y - getHeight() / 2);  
 }  
 if (finishedLevel) {  
 float barrier = 64 / MarioBros.*PPM*;  
 float b2bodyPosition\_Y = b2body.getPosition().y;  
  
 if (currentState == State.*STANDING* && b2bodyPosition\_Y < barrier) {  
 b2body.setActive(false);  
 }  
 }  
 setRegion(getFrame(dt));  
 if (timeToDefineBigMario) {  
 defineBigMario();  
 }  
 if (timeToRedefineMario) {  
 redefineMario();  
 }  
}

Metoda update klasy PlayScreen gdzie aktualizowane są wszystkie logiczne obiekty w grze.

public void update(float dt) {  
 handleInput(dt);  
 handleSpawningItems();  
 world.step(1 / 60f, 6, 2);  
 player.update(dt);  
 for (Enemy enemy : creator.getEnemies()) {  
 enemy.update(dt);  
 if (enemy.getX() < player.getX() + 224 / MarioBros.*PPM*) {  
 enemy.b2body.setActive(true);  
 }  
 }  
 for (Item item : items) {  
 item.update(dt);  
 }  
 hud.update(dt);  
 if (player.currentState != Mario.State.*DEAD*) {  
 if (player.b2body.getPosition().x >= gamecam.position.x) {  
 gamecam.position.x = player.b2body.getPosition().x;  
 }  
 }  
 gamecam.update();  
 renderer.setView(gamecam);  
}

|  |  |  |  |
| --- | --- | --- | --- |
| Tabela oceny projektu z Programowania urządzeń mobilnych | | | |
| lp | Oceniany element | max | punkty |
| 1 | Ogólny opis gry (wizja) | 1 |  |
| 2 | Analiza dziedziny | 2 |  |
| 3 | Specyfikacja wymagań | 2 |  |
| 4 | Analiza i projekt: |  |  |
|  | Architektura systemu gry | 2 |  |
|  | Obiektowy model analizy gry | 2 |  |
|  | Projekt oprogramowania | 2 |  |
| 5 | Czytelność kodu źródłowego | 2 |  |
| 6 | Podział kodu na moduły/biblioteki/klasy | 1 |  |
| 7 | Zgodność gry z oryginałem | 6 |  |
| 8 | Dźwięki | 4 |  |
| 9 | Muzyka | 1 |  |
| 10 | Sterowanie/nawigacja | 2 |  |
| 11 | Dodatkowa wersja z rozbudowaną grafiką, dźwiękiem, muzyką | 5 |  |
| 12 | Inwencja własna | 3 |  |
| 13 | Punkty przyznane od prowadzącego | 3 |  |
|  |  | 40 |  |