Sprawozdanie Kamień milowy 2 :

1. Większość prezentacji znajduje się w formie wideo :
2. Zakres prac zrealizowanych :

* Adrian Kunecki:

Stworzenie grafik, mapy, tekstur oraz ich spakowanie.

Implementacja Kolizji

Stworzenie Klasy Ship, Main oraz Szkieletu B2WordlCreator

Poprawienie wizualne HUD i Menu

Tworzenie zapisu odczytu i oprogramowanie działania przycisków

* Piotr Lepa:

Stworzenie klasy EnemyShip, EnemyBlaster, Blaster

Stworzenie modelu poruszania się Obiektów

Implementacja systemu odświeżenia list obiektów

Poprawienie gameplayu (poruszanie się statku gracza, modyfikatory rozgrywki, poruszanie się fal przeciwników rozbudowa pliku Tiled i porusznia się tła gry)

* Prace wspólne:

Rozwój i implementacja metod i funkcji w klasie PlayScreen

Stworzenie MenuScreen i rozwój

1. Link do sklonowania projektu z gitHub:

https://github.com/ProjektyPSK/ProjecktGKLibGDX.git

1. Tematy nieomówione w prezentacji

* Zasada działania upgrade’ów:

Jeśli przeciwnik zostanie trafiony jest 40% szansa że wypadnie z niego upgrade (o ile poprzedni został już usunięty)

else if (20 > rand.nextInt(50))  
 *upgrade* = new Upgrade(world, screen , this.getX() + this.getWidth() / 2, this.getY());

Jeśli statek gracza dotknie go jego szybkość strzału zostanie zwiększona, jeśli osiągnie zbyt wysoką wartość zmieni, się jego patern strzału (podwójn, potem potrójny), jeśli zostanie osiągnięty maksymalny typ strzału za każdy następny zebrany upgrade zostanie zwiększony wynik gracza.

public void colideWithUpgrade (){  
 if(shootTime > 0.2f) {  
 shootTime \*= 0.85;  
 }  
 else if (shootType < 3){  
 shootType ++;  
 }  
 else {  
 Hud.*updateScore*(300);  
 }  
  
}

Jeśli dojdzie do kolizji gracza z przeciwnikiem lub pociskiem dojdzie do sytuacji odwornej do opisanej powyżej, tempo strzału zostanie zmniejszone lub rodzaj strzały ulegnie zmianie.

public void colideWithEntiti() {  
 if (this.lives > 1) {  
 this.lives--;  
 Hud.*updateLives*(-1);  
 Hud.*updateScore*(-300);  
 if(shootTime > 0.2f) {  
 shootTime \*= 1.15;  
 }  
 else if (shootType > 1){  
 shootType --;  
 }  
 else {  
 shootTime \*= 1.15;  
 }

* Zasada działania tła:

Statek gracza w grze nie porusza się w ogóle w osi Y za to tło znajdujące się za nim porusza się cały czas, tło składa się z 12 elementów, porószających się z różną prędkością, wartwy tła dzielą się na skrajne, środkowe i wewnętrzne, każda z nich porusza się coraz szybciej co symuluje wrażenie szybkiego poruszania się. Elementy znajdujące się ”bliżej” (perspektywa) ekrany poruszają się wolniej a te dalej szybciej, rozwiązane będzie poprawne dla wielu rodzajów obrazów, więc jest ono uniwersalne i umożliwia wprowadzenie dowolnych plików .png.

Szybkośc poruszania się warstw:

switch (layer) {  
 case 0 :  
 this.b2body.applyLinearImpulse(new Vector2(0, -1.0f), this.b2body.getWorldCenter(), true);  
 break;  
 case 1 :  
 this.b2body.applyLinearImpulse(new Vector2(0, -2.0f), this.b2body.getWorldCenter(), true);  
 break;  
 case 2 :  
 this.b2body.applyLinearImpulse(new Vector2(0, -3.0f), this.b2body.getWorldCenter(), true);  
 break;

Ułożenie wartw i ich wielkość:

switch (layer) {  
 case 0 :  
 setBounds(x / Main.*PPM*, y / Main.*PPM*, 490 / Main.*PPM*, 1750 / Main.*PPM*);  
 break;  
 case 1 :  
 setBounds(x / Main.*PPM*, y / Main.*PPM*, 350 / Main.*PPM*, 1750 / Main.*PPM*);  
 break;  
 case 2 :  
 setBounds(x / Main.*PPM*, y / Main.*PPM*, 210 / Main.*PPM*, 1750 / Main.*PPM*);  
 break;

Wysokośc jednego obiektu to 1750 px, bazowa wysokośc ekranu to 1080px, więc jeśli którykolwiek obiekt tła przekroczy wartość -1750 px (dolna krawędź) ma pojawić się na pozycji +1747px. 3px zapasu zostały zastosowane w celu uniknięcia przestrzeni między obiektami w przypadku odświerzenia nierównego z klatkami gry.

public void backgroundChecker (){  
 if(background.size() > 0){  
 for (Background bac : background){  
 if (bac.b2body.getTransform().getPosition().y < -17.5f){  
 bac.b2body.setTransform(bac.b2body.getPosition().x, 17.47f, 0);  
 }  
 }  
 }  
}

* Pulsowanie pocisków

Ten element projektu został wykonany w dosyć prosty sposób, a uzyskany efekt wizualny jest przyjemny dla oka

if(growShrink) {  
 size++;  
 setBounds(b2body.getPosition().x - getWidth() /2,b2body.getPosition().y - getHeight() /2, size / Main.*PPM*, size / Main.*PPM*);  
 }  
 if(!growShrink) {  
 size--;  
 setBounds(b2body.getPosition().x - getWidth() /2,b2body.getPosition().y - getHeight() /2, size / Main.*PPM*, size / Main.*PPM*);  
 }  
 if (size > 30)  
 growShrink = false;  
 if(size < 10)  
 growShrink = true;  
  
 }

* Masa obiektów

W bibliotece libGDX każdy obiekt posiadający ciało (b2body) posiada domyślną mase równą ‘1’.

Jeśli dojdzie do kolizji dwóch obiektów (oba muszą posiadać wzajemnie swoje maski botowe [omawiane w poprzednim filmie na YT]) to obiekty zostaną odepchnięte od siebie z zachowaniem zasad nadaj masy.

W przypadku większości kolizji zastosownych w aplikacji, conajmniej jeden z obiektów jest usuwany, jednak może dojść do tego, że przed usunięciem obiektu, zdążą się od siebie odbić co sprawia, że statek gracza lub przeciwnik, mogliby być przesunięci za mapę, by tego uniknąć większość mas obiektów jest znacząco zmiejszana w przypadku ryzyka kolizji (pociski, nurkujące statki)

MassData mass = new MassData();  
mass.mass = 0.0001f;  
b2body.setMassData(mass);

* Poruszanie się statków przeciwnika

Zaprezentowana funkcja jest dosyć skomplikowana dlatego opis jej zostanie podzielony na części.

* Obecnie istnieją 3 paterny poruszania się przeciwników, każdy z nich jest nadawany w momencie inicjalizacji fali.

W zależności od paterny statki powinny kolidować, z różnymi obiektami, dlatego już w konstruktorze jest to definiowane

if (movementType == 0)  
fdef.filter.maskBits = Main.*BORDER* | Main.*SHIP\_ENEMY\_BIT* | Main.*SHIP\_HERO\_BIT* | Main.*BLASTER\_HERO*;  
else if (movementType == 1)  
 fdef.filter.maskBits = Main.*SHIP\_ENEMY\_BIT* | Main.*SHIP\_HERO\_BIT* | Main.*BLASTER\_HERO*;  
else if (movementType == 2)  
 fdef.filter.maskBits = Main.*SHIP\_HERO\_BIT* | Main.*BLASTER\_HERO*;  
fdef.shape = shape;  
b2body.createFixture(fdef).setUserData(this);

* Pierwszy styl poruszania się:

Gdy statek zleci na odpowiednią wysokość rozpoczyna się jego przypadkowy ruch zaprezentowany w podczas omawiania kamienia milowego 2:

if (movementTime >= (rand.nextFloat() + 0.5f) && !diving) {  
 if (Math.*abs*(player.getY() - this.getY()) > 8.5f)  
 this.b2body.setLinearVelocity(new Vector2(0, -1));  
 else if (Math.*abs*(player.getY() - this.getY()) < 0.5f)  
 this.b2body.setLinearVelocity(new Vector2(0, 1));  
 else {  
 float randX = (rand.nextFloat() - 0.5f);  
 float randY = (rand.nextFloat() - 0.5f);  
  
 this.b2body.setLinearVelocity(new Vector2(randX, randY));  
 }  
 movementTime = 0;  
}

ruch ten został wzbogacony o opcje nurkowania (warunek diving widoczny w powyższej funcji “if”)

Każdy statek indywidualnie wybiera moment zanurkowania na podstawie generatora liczb losowych

if (diveTime >= (rand.nextInt(10000)) + 5) {  
 if (!diving) {  
 this.b2body.setLinearVelocity(new Vector2(0, -2));  
 b2body.destroyFixture(b2body.getFixtureList().get(0));  
  
 FixtureDef fdef = new FixtureDef();  
 CircleShape shape = new CircleShape();  
 shape.setRadius(35 / Main.*PPM*);  
  
 fdef.filter.categoryBits = Main.*SHIP\_ENEMY\_BIT*;  
 fdef.filter.maskBits = Main.*BORDER* | Main.*SHIP\_HERO\_BIT* | Main.*BLASTER\_HERO*;  
 fdef.shape = shape;  
 b2body.createFixture(fdef).setUserData(this);  
 MassData mass = new MassData();  
 mass.mass = 0.0001f;  
 b2body.setMassData(mass);  
 }  
 diving = true;  
}

W przypadku rozpoczęcia manewru nurkowania jest zmieniana masa obiektu oraz ustawiane są nowe maskiBitów kolizji, które sprawią że statek nie będzie kolidował z innymy przeciwnikami, pominięcie tego warunku, mogłoby sprawić nieporządany ruch statku w trakcie manewru nurkowania jak i pozostałych przciwników.

Jeśli dojdzie do przekroczenia przez statek poziomu -100 px (promień statku to 35 px) pojawia się on na górze ekrany i są mu przywracane parametry początkowe

if (b2body.getTransform().getPosition().y < -1.0f) {  
 b2body.setTransform(b2body.getPosition().x, 12f, 0);  
 diving = false;  
  
 b2body.destroyFixture(b2body.getFixtureList().get(0));  
  
 FixtureDef fdef = new FixtureDef();  
 CircleShape shape = new CircleShape();  
 shape.setRadius(35 / Main.*PPM*);  
  
 fdef.filter.categoryBits = Main.*SHIP\_ENEMY\_BIT*;  
 fdef.filter.maskBits = Main.*BORDER* | Main.*SHIP\_ENEMY\_BIT* | Main.*SHIP\_HERO\_BIT* | Main.*BLASTER\_HERO*;  
 fdef.shape = shape;  
 b2body.createFixture(fdef).setUserData(this);  
 MassData mass = new MassData();  
 mass.mass = 1f;  
 b2body.setMassData(mass);  
}

* Drugi styl porusznia się:

Został on stworzony na wzór starych gier typu space invaders stateki poruszają się w lewo, w dół, w prawo, w dół, w lewo ......

if(movementType == 1){  
 if(movementTime > 1f && moveRight && canShoot) {  
 this.b2body.setLinearVelocity(new Vector2(1f, 0));  
 moveRight = false;  
 moveDown = true;  
 movementTime = 0;  
 }  
 if(movementTime > 2f && moveDown) {  
 moveLeftOrRight ++;  
 this.b2body.setLinearVelocity(new Vector2(0, -1f));  
 if(moveLeftOrRight == 1) {  
 moveRight = true;  
 moveDown = false;  
 }  
 if(moveLeftOrRight == 2){  
 moveLeft = true;  
 moveDown = false;  
 moveLeftOrRight = 0;  
 }  
 movementTime = 0;  
 }  
 if(movementTime > 1f && moveLeft) {  
 this.b2body.setLinearVelocity(new Vector2(-1f, 0));  
 moveLeft = false;  
 moveDown = true;  
 movementTime = 0;  
 }  
 if (b2body.getTransform().getPosition().y < -1.0f) {  
 b2body.setTransform(b2body.getPosition().x, 11f, 0);  
 }  
}

* Ostatni styl poruszania się:

Polega na opuszczeniu fali na odpowiednią wysokośc po czym jej jednostajny ruch w prawu, w przypadku przekroczenia krawędzi ekranu, pojawiają się na jego przeciwległej stronie

if(movementType == 2) {  
 if (movementTime < 7f)  
 this.b2body.setLinearVelocity(new Vector2(0, -1));  
 else {  
 this.b2body.setLinearVelocity(new Vector2(2.5f, 0));  
 if (b2body.getTransform().getPosition().x > 21.5f) {  
 b2body.setTransform(-0.5f, b2body.getPosition().y, 0);  
 }  
 }  
}

* Hud:

Jest obsługiwany w dwóch formach, dla PlayScreen oraz dla MenuScreen.

Dla PlayScreen odlicza on pozostąły czas i ustawia wartość wyniku.

Dla ManuScreen w przypadku zakończenia gry lub przegranej przelicza czas na wynik

public void update (float dt){  
 if(tryb ==1) {  
 timeCount += dt;  
 if (timeCount >= 1) {  
 *worldTimer*--;  
 countdownLabel.setText(String.*format*("%03d", *worldTimer*));  
 timeCount = 0;  
 }  
 }  
 if(tryb == 2){  
 if( *worldTimer* > 0){  
 *updateScore*(5);  
 *worldTimer* --;  
 countdownLabel.setText(String.*format*("%03d", *worldTimer*));  
 }  
 if( *worldTimer* < 0){  
 *updateScore*(-5);  
 *worldTimer* ++;  
 countdownLabel.setText(String.*format*("%03d", *worldTimer*));  
 }