Министерство Образования Республики Молдова

Технический Университет Молдовы

Кафедра Автоматики и Информационных Технологий

Лабораторная работа №5

По дисциплине: «MIDPS»

Тема: «Interactive Development Environments Laboratory»

Выполнил: студент группы TI-145: Русу Дмитрий

Проверила: Кожану Ирина

Кишинёв 2016

Цель работы:

O alta aplicatie sofisticata la alegere

- Game

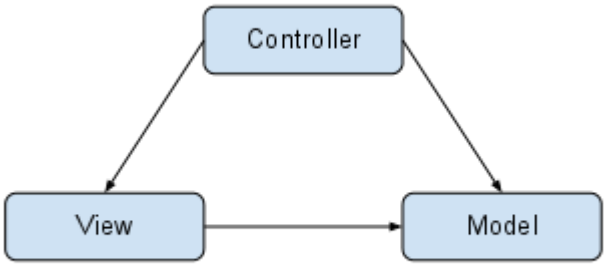
Теоретическая часть:

После рассмотрения жизненного цикла игры сразу стоит рассмотреть архитектуру (каркас). Вообще Роллингс и Моррис (Rollings and Dave Morris) в своей книге “Game Architecture and Design” подробно описывают создание игр с точки зрения архитектуры. В своё время я правда не особо проникся этой книгой, но вам может понравится. Я же опишу архитектуру, которую стараюсь использовать сам.

Разбиение приложения на компоненты со слабым связыванием – это не просто какой-то идеологических ход, такой подход действительно очень упрощает разработку. В частности, я предлагаю использовать заезженный паттерн – **MVC**. Часть материала брал с занятного сайта http://obviam.net/. Там вообще очень много полезной информации для разработчиков игр.

MVC

Довольно-таки удобный образец архитектуры для разработки игр. Главным его преимуществом, как по мне, является то, что можно вносить изменения в какую-то часть игры, при этом не затрагивая остальные компоненты приложения.



Примерно как всё в играх происходит? Игрок производит какую-то манипуляцию:

1. Игрок нажимает на экран (или на клавиатуру).
2. В **controller** обрабатывается нажатие. Здесь же по сути вся логика реализована: проверка на препятствия, отслеживание состояний объектов, изменение их состояний и т.д.
3. То есть, **controller** изменяет состояние объектов (**model**‘s).
4. После чего объекты отрисовываются (**view**).

MVC очень удачно подходит. Если ещё не поняли, поясню кое-какие моменты. Объекты (**Model**) абсолютно ничего не знают про рендеринг. Многие пишут, что объекты не должны и состояние менять сами, а за них это должен делать контроллер. Я к этому вопросы подошёл практически. Возьмите, к примеру, вашего персонажа, которому надо как-то описать логику обхода препятствий. Большинство скажет, что в контроллере сие дело надо реализовывать. Но почему? Ведь, когда вы идёте по улице, то обходите препятствие после собственных расчётов, а не мир или контроллер просчитывает это дело. Так что, часть логики взаимодействия с миром я бы посоветовал именно в сами объекты добавлять.

Я имею ввиду именно живые объекты (если можно так сказать про виртуальных персонажей (:). Логику неодушевлённых предметов можно и в контроллере делать. Перейдём к практической части. За основу берём проект изведения.

Игровые компоненты

В данной статье покажу как разбить на части приложение. Немного про объекты мира расскажу.

Создание мира и его объектов

Добавьте к проекту package **suvitruf.libgdxtutorial.model**. Здесь у нас будут объекты мира. Добавьте в этот пакет класс `Brick` и зададим базовые свойства:
view sourceprint

```
01.package suvitruf.libgdxtutorial.model;
02.
03.//импорт нужных либ
04.import com.badlogic.gdx.math.Rectangle;
05.import com.badlogic.gdx.math.Vector2;
06.
07.public class Brick {
08.//размер объекта
09.static final float SIZE = 1f;
10.//координаты
11.Vector2 position = new Vector2();
12.Rectangle bounds = new Rectangle();
13.
14.public Brick(Vector2 pos) {
15.this.position = pos;
16.this.bounds.width = SIZE;
17.this.bounds.height = SIZE;
18.}
19.
20.public Rectangle getBounds() {
21.return bounds;
22.}
23.
24.public Vector2 getPosition() {
25.return position;
26.}
27.}
```

У блока нет никакой логики, он представляет собой...ммм...просто кирпич. Он ни с чем не взаимодействует, но с ним могут взаимодействовать живые объекты. Мы используем тип `Vector2` от libgdx. Это позволяет нам работать лишь с Евклидовыми векторами. Мы будем

использовать векторы для позиционирования, вычисления скорости и для движения (ну да, кирпич не двигается...но наш персонаж будет).

Далее добавим класс (добавьте класс `Player` к пакету**suvitruf.libgdxtutorial.model**), который будет являть собой нашего персонажа.

view sourceprint

```
01.package suvitruf.libgdxtutorial.model;
02.
03.import com.badlogic.gdx.math.Rectangle;
04.import com.badlogic.gdx.math.Vector2;
05.
06.public class Player {
07.
08.//состояние
09.public enum State {
10.NONE, WALKING, DEAD
11.}
12.
13.
14.//скорость движения
15.public static final float SPEED = 2f;
16.//размер
17.public static final float SIZE = 0.7f;
18.
19.//позиция в мире
20.Vector2 position = new Vector2();
21.//используется для вычисления движения
22.Vector2 velocity = new Vector2();
23.//прямоугольник, в который вписан игрок
24.//будет использоваться в будущем для нахождения коллизий (столкновение со стенкой
и т.д.
25.Rectangle     bounds = new Rectangle();
26.//текущее состояние
27.State     state = State.NONE;
28.
29.public Player(Vector2 position) {
30.this.position = position;
31.this.bounds.height = SIZE;
32.this.bounds.width = SIZE;
33.}
34.
35.public Rectangle getBounds() {
36.return bounds;
37.}
38.
39.public Vector2 getVelocity() {
40.return velocity;
41.}
42.
43.public Vector2 getPosition() {
44.return position;
45.}
46.
47.//обновления движения
48.public void update(float delta) {
49.position.add(velocity.tmp().mul(delta));
50.}
51.}
```

Теперь нам нужно создать мир, в котором будут все эти объекты. Добавляем в пакет**suvitruf.libgdxtutorial.model** класс `World`. Мир условно делится на клетки. К примеру создадим мир 8×5.

view sourceprint

```
01.package suvitruf.libgdxtutorial.model;
02.
03.
04.import com.badlogic.gdx.math.Vector2;
05.import com.badlogic.gdx.utils.Array;
06.
07.public class World {
08.//массив блоков
09.Array<Brick> bricks = new Array<Brick>();
10.//наш персонаж
11.public Player player;
12.
13.//ширина мира
14.public int width;
15.//высота мира
16.public int height;
17.
18.//получить массив блоков
19.public Array<Brick> getBricks() {
20.return bricks;
21.}
22.//получить игрока
23.public Player getPlayer() {
24.return player;
25.}
26.
27.public World() {
28.width = 8;
29.height = 5;
30.createWorld();
31.}
32.
33.//создадим тестовый мир какой-нибудь
34.public void createWorld() {
35.player = new Player(new Vector2(6,2));
36.bricks.add(new Brick(new Vector2(0, 0)));
37.bricks.add(new Brick(new Vector2(1, 0)));
38.bricks.add(new Brick(new Vector2(2, 0)));
39.bricks.add(new Brick(new Vector2(3, 0)));
40.bricks.add(new Brick(new Vector2(4, 0)));
41.bricks.add(new Brick(new Vector2(5, 0)));
42.bricks.add(new Brick(new Vector2(6, 0)));
43.bricks.add(new Brick(new Vector2(7, 0)));
44.
45.
46.}
47.}
```

`World` – модель мира. По сути он является контейнером для объектов, что логично (:

Контроллер

Создадим пакет **suvitruf.libgdxtutorial.controller** и добавим в него класс`WorldController`. В этом классе как раз и будет реализована логика вся. По идеи в контроллере производятся

изменения состояний объектов мира. И главное, в зависимости от действий юзера будут манипуляции с объектом `Player`.

view sourceprint

```
001.package suvitruf.libgdxtutorial.controller;
002.
003.import java.util.HashMap;
004.import java.util.Map;
005.import suvitruf.libgdxtutorial.model.*;
006.
007.public class WorldController {
008.
009.//направление движения
010.enum Keys {
011.LEFT, RIGHT, UP, DOWN
012.}
013.//игрок
014.public Player player;
015.
016.//куда движемся...игрок может двигаться одновременно по 2-м направлениям
017.static Map<Keys, Boolean> keys = newHashMap<WorldController.Keys, Boolean>();
018.
019.//первоначально стоим
020.static {
021.keys.put(Keys.LEFT, false);
022.keys.put(Keys.RIGHT, false);
023.keys.put(Keys.UP, false);
024.keys.put(Keys.DOWN, false);
025.};
026.
027.public WorldController(World world) {
028.this.player = world.getPlayer();
029.}
030.
031.//флаг устанавливаем, что движемся влево
032.public void leftPressed() {
033.keys.get(keys.put(Keys.LEFT, true));
034.}
035.
036.//флаг устанавливаем, что движемся вправо
037.public void rightPressed() {
038.keys.get(keys.put(Keys.RIGHT, true));
039.}
040.
041.//флаг устанавливаем, что движемся вверх
042.public void upPressed() {
043.keys.get(keys.put(Keys.UP, true));
044.}
045.
046.//флаг устанавливаем, что движемся вниз
047.public void downPressed() {
048.keys.get(keys.put(Keys.DOWN, true));
049.}
050.
051.//освобождаем флаги
052.public void leftReleased() {
053.keys.get(keys.put(Keys.LEFT, false));
054.}
055.public void rightReleased() {
056.keys.get(keys.put(Keys.RIGHT, false));
057.}
```

```
058.public void upReleased() {
059.keys.get(keys.put(Keys.UP, false));
060.}
061.public void downReleased() {
062.keys.get(keys.put(Keys.DOWN, false));
063.}
064.
065.//главный метод класса...обновляем состояния объектов здесь
066.public void update(float delta) {
067.processInput();
068.player.update(delta);
069.}
070.
071.public void resetWay(){
072.rightReleased();
073.leftReleased();
074.downReleased();
075.upReleased();
076.}
077.
078.//в зависимости от выбранного направления движения выставляем новое направление
движения для персонажа
079.private void processInput() {
080.if (keys.get(Keys.LEFT))
081.player.getVelocity().x = -Player.SPEED;
082.
083.if (keys.get(Keys.RIGHT))
084.player.getVelocity().x =Player.SPEED;
085.
086.if (keys.get(Keys.UP))
087.player.getVelocity().y = Player.SPEED;
088.
089.
090.if (keys.get(Keys.DOWN))
091.player.getVelocity().y = -Player.SPEED;
092.
093.//если не выбрано направление, то стоим на месте
094.if ((keys.get(Keys.LEFT) && keys.get(Keys.RIGHT)) || (!keys.get(Keys.LEFT) &&
(!keys.get(Keys.RIGHT))))
095.player.getVelocity().x = 0;
096.if ((keys.get(Keys.UP) && keys.get(Keys.DOWN)) || (!keys.get(Keys.UP) &&
(!keys.get(Keys.DOWN))))
097.player.getVelocity().y = 0;
098.
099.}
100.}
```

Рендеринг

И так, про контроллер и объекты мира поговорили. Теперь нужно отрисовать объекты наши.

Для этого создадим пакет **suvitruf.libgdxtutorial.view**, а в нём класс `WorldRenderer`.

view sourceprint

```
01.package suvitruf.libgdxtutorial.view;
02.
03.import suvitruf.libgdxtutorial.model.*;
04.import com.badlogic.gdx.graphics.Color;
05.import com.badlogic.gdx.graphics.OrthographicCamera;
06.import com.badlogic.gdx.graphics.glutils.ShapeRenderer;
07.import com.badlogic.gdx.graphics.glutils.ShapeRenderer.ShapeType;
08.import com.badlogic.gdx.math.Rectangle;
```

```
09.
10.public class WorldRenderer {
11.public static float CAMERA_WIDTH = 8f;
12.public static  float CAMERA_HEIGHT = 5f;
13.
14.private World world;
15.public OrthographicCamera cam;
16.ShapeRenderer renderer = new ShapeRenderer();
17.
18.
19.public int width;
20.public int height;
21.public float ppuX;    // пикселей на точку мира по X
22.public float ppuY;    // пикселей на точку мира по Y
23.
24.public void setSize (int w, int h) {
25.this.width = w;
26.this.height = h;
27.ppuX = (float)width / CAMERA_WIDTH;
28.ppuY = (float)height / CAMERA_HEIGHT;
29.}
30.//установка камеры
31.public void SetCamera(float x, float y){
32.this.cam.position.set(x, y,0);
33.this.cam.update();
34.}
35.
36.public WorldRenderer(World world) {
37.
38.this.world = world;
39.this.cam = new OrthographicCamera(CAMERA_WIDTH, CAMERA_HEIGHT);
40.//устанавливаем камеру по центру
41.SetCamera(CAMERA_WIDTH / 2f, CAMERA_HEIGHT / 2f);
42.
43.}
44.
45.//основной метод, здесь мы отрисовываем все объекты мира
46.public void render() {
47.drawBricks();
48.drawPlayer() ;
49.
50.}
51.
52.//отрисовка кирпичей
53.private void drawBricks() {
54.renderer.setProjectionMatrix(cam.combined);
55.//тип устанавливаем...а данном случае с заливкой
56.renderer.begin(ShapeType.FilledRectangle);
57.//прогоняем блоки
58.for (Brick brick : world.getBricks()) {
59.Rectangle rect =  brick.getBounds();
60.float x1 =  brick.getPosition().x + rect.x;
61.float y1 =  brick.getPosition().y + rect.y;
62.renderer.setColor(new Color(0, 0, 0, 1));
63.//и рисуем блоки
64.renderer.filledRect(x1, y1, rect.width, rect.height);
65.}
66.
67.renderer.end();
68.}
```

```
69.
70.//отрисовка персонажа по аналогии
71.private void drawPlayer() {
72.renderer.setProjectionMatrix(cam.combined);
73.Player player = world.getPlayer();
74.renderer.begin(ShapeType.Rectangle);
75.
76.Rectangle rect = player.getBounds();
77.float x1 = player.getPosition().x + rect.x;
78.float y1 = player.getPosition().y + rect.y;
79.renderer.setColor(new Color(1, 0, 0, 1));
80.renderer.rect(x1, y1, rect.width, rect.height);
81.renderer.end();
82.}
83.
84.}
```

ppuX и **ppuY** очень важны…Ведь мир у нас 8на5, а экран телефона в пикслеях не соответствует этим размерам. Поэтому нужны эти переменные, которые при рендеринге будут корректировать координаты объектов для вывода на реальный экран телефона.

OrthographicCamera cam – камера, которая используется для того, чтобы “посмотреть” на мир. В текущем примере мир очень маленький, и он влезает в камеру, но когда у нас будет большой уровень, и персонаж будет перемещается в нем, мы должны будем менять положение камеры. Собственно, там и расчёт координат изменится…В будущих статьях остановлюсь на этом.

GameScreen

Теперь осталось лишь связать все наши компоненты вместе. Для этого создадим пакет **suvitruf.libgdxtutorial.screens**, а в нём класс `GameScreen`.

GameScreen реализует интерфейс Screen, который очень походит на ApplicationListener, но у этого есть 2 важных ключевых отличия (два метода). **show()** – вызывается, когда становится активным. **hide()**– вызывается, когда активным становится другой экран.