Panca Game Maker – DOC

Diogo Fernando Trevisan

0.1 eqs

$\frac{\partial E(n)}{\partial W_{kj}(n)}$

CHAPTER

1

TILESET

1.1 Introduction

The tileset is a set of *Tiles*. Each Tileset is built using an image. This image is divided into a set of subimages, each of this subimage is a *Tile*. All Tiles in a tileset are square and have the same size (pixels).

1.2 Tile

Each tile of a Tileset as attributes one image (java.awt.Image) and informations about wich directions the player and enemies can pass in this tile, as shows Figure 1.1.

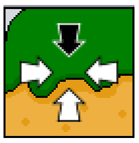


Figure 1.1: Tile: a tile have information about wich directions the player can pass. In this example the player can't pass from up to down.

1.3 Tileset

A tileset is a set of tiles. An image is loaded and a tile size is set. The image of Tileset is cut into as many as possible Tiles. This set of tiles are used to create an Scenary. Figure 1.2 shows an tileset example.

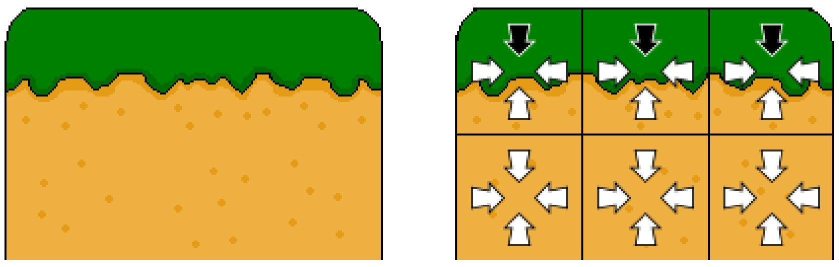


Figure 1.2: Tileset: the image (a) is cut into various subimages. Each subimage is a *Tile* and have its own attributes.

1.4 File Format

The tileset can be saved to a file. It is written in binary format, and the format is shown in Listing 1.

Listing 1 Tileset File Format.

```
$[int] Number\_tiles\_Y$
$[int] Number\_tiles\_X$
$[int] Tile\_size$
**for** $i = 0 \ldots Number\_tiles\_Y$ **do**
  **for** $j = 0 \ldots Number\_tiles\_X$ **do**
    $[int] tile[i][j].terrain\_type$
    $[bool] tile[i][j].pass\_from\_up$
    $[bool] tile[i][j].pass\_from\_down$
    $[bool] tile[i][j].pass\_from\_left$
    $[bool] tile[i][j].pass\_from\_right$
    $[int] tile[i][j].image\_height$
    $[int] tile[i][j].image\_width$
    **for** $x = 0 \ldots tile[i][j].image\_width$ **do**
      **for** $y = 0 \ldots tile[i][j].image\_height$ **do**
        $[int] tile[i][j].image[x][y]$
      **end for**
    **end for**
  **end for**
**end for**
```

CHAPTER

2

GRAPHICS

2.1 Sprite

Each Sprite consists of an image. The sprite have a display time (in seconds, double), and one collision box with scenary. If none collision box is provided, the entire sprite is a collision box as shows Figure 2.1.

Each sprite may have a set of collision boxes where is vulnerable and a set of collision boxes where is attacking.

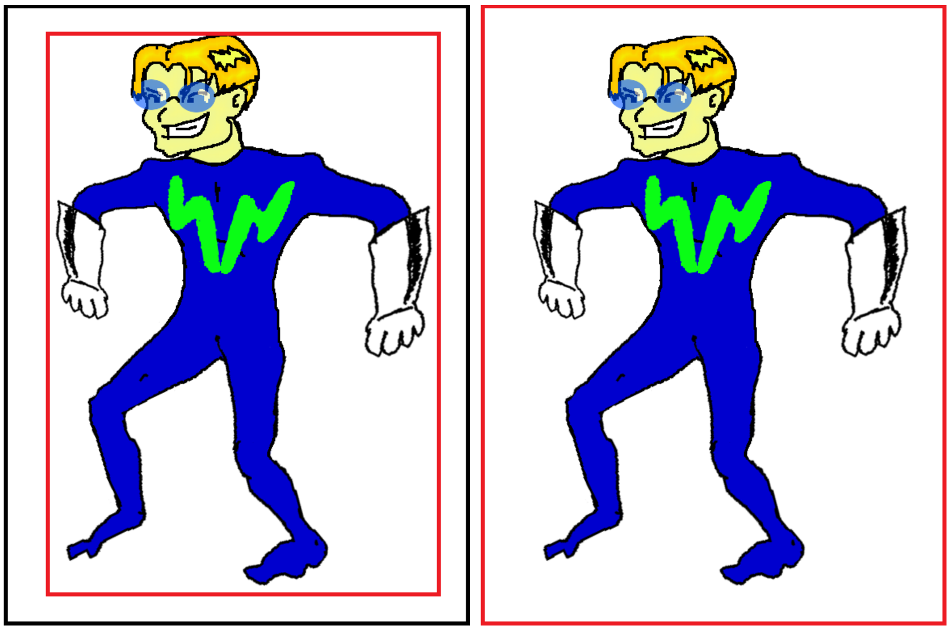


Figure 2.1: Sprite with an collision box (left, in red), and without defining an collision box (right, red).

2.1.1 Drawing the Sprite

All Sprite are drawin from the bottom of the point (MIDDLE, 0). The position of the character marks it position and the character is drawn based in this position as shows Figure 2.2.

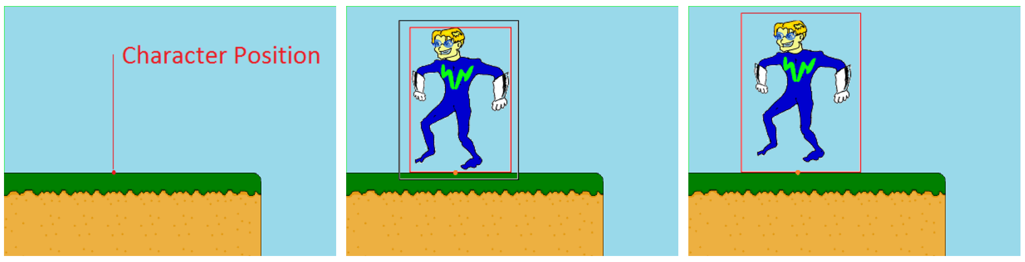


Figure 2.2: Positioning the sprite in the character position.

Thus, we can see that it is recommended to always make all sprite animation centering the sprites, and make all character and enemies sprites with the same size.

2.1.2 Sprite Coordinate System

The Sprite coordinate system is inverse to the bitmaps coordinate system. The (0,0) point is the lower left corner and the (W,H) point is the upper right point as shows Figure **??**.

2.2 Window & Drawing

The game can support multiple resolutions. Before draw the game to the computer screen it is drawn to a buffer. **ALL GRAPHIC DATA MUST BE CREATED USING THE INTERNAL RESOLUTION**. Table **??** shows the recommended size of graphics and the size of internal window. All data is drawn to the buffer and the buffer is resized in order to fill the screen as can be seen in Figure 4.2.

2.3 Rendering

The Interface *render.Renderable* is used in the *render.Render* class. This Interface has one method *render(Graphics2D g)* wich takes one argument. This argument is the draw buffer in the size of the internal window. All objects that will be rendered must implement this method. In the GameManager these objects must be put in the render queue – the Render access the *render* method of all objects in depth order. The usuar order to draw is Background, Scenary, character (playabe and enemies) and items, and, at last the Head Up Display (HUD).

CHAPTER

3

BACKGROUND

3.1 Introduction

3.2 Background Layer

Each background is composed of several layers. Each layer has it's own properties. There are some types of background layers:

- **ColorBackgroundLayer:** A simple layer that fills the screen with a solid color;

- **ImageBackGroundLayer:** A layer that fills the screen with a image. The image can be stretched to fill the screen or can be tiled horizontally, vertically or in both directions.

3.3 File Format

CHAPTER

4

SCENARY

4.1 Introduction

Each Scenary is built using a *Tileset*. A Scenary have a number Y of vertical cells and a number X of horizontal cells. Each cell is a reference (pointer) to an *Tile* of the Scenary *Tileset*.

4.2 The Scenary

A Scenary is a matrix of Tiles (see 1) called cells. The cells starts from 0 – height, and 0 – width. The first cell is upper left corner, as shows fig 4.1. Each *Scenary* has a tileset, a *Background* and a visualisation window of x by y pixels wich moves across the scenary and try to center the character.

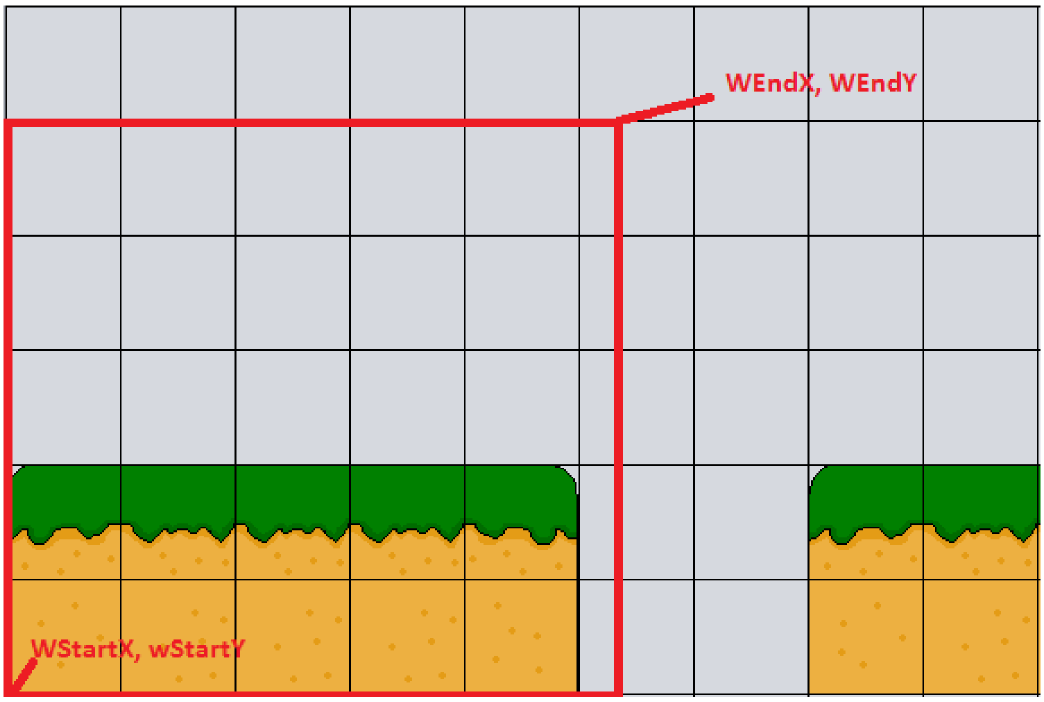


Figure 4.1: The scenary have a “moving window” next to character.

In game, all scenaries have the same tile size and the same window size. The window is computed using the internal size. If an scenary have 10×200 tiles and each tile have 32 pixels, the scenary size will be 320×6400 pixels, thus, the window size should consider this dimensions. Each window will be resized to fit the *screen resolution* as show Figure 4.2.

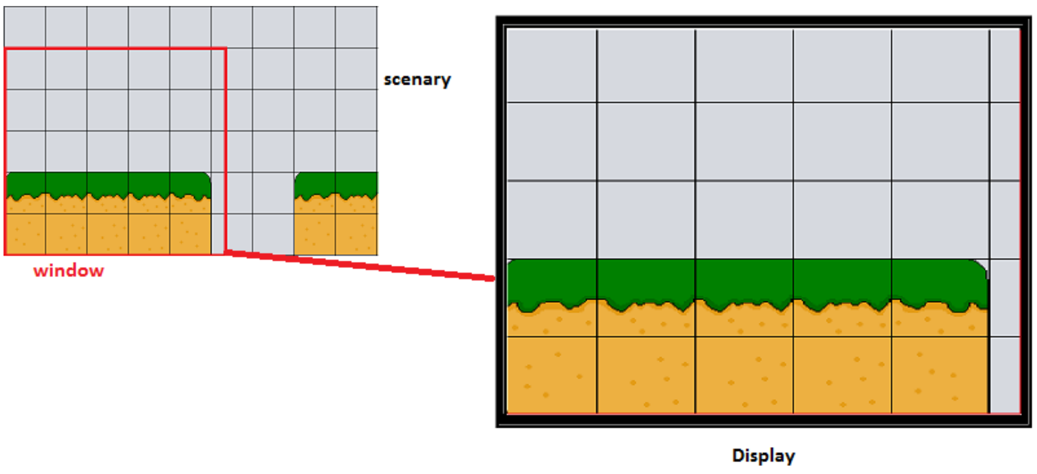


Figure 4.2: The buffer is resized to fill up the screen.

And the scenary coordinate system starts with (X,Y) – at left bottom, and ends with (X,Y) – at right upper, as shows Figure **??**.

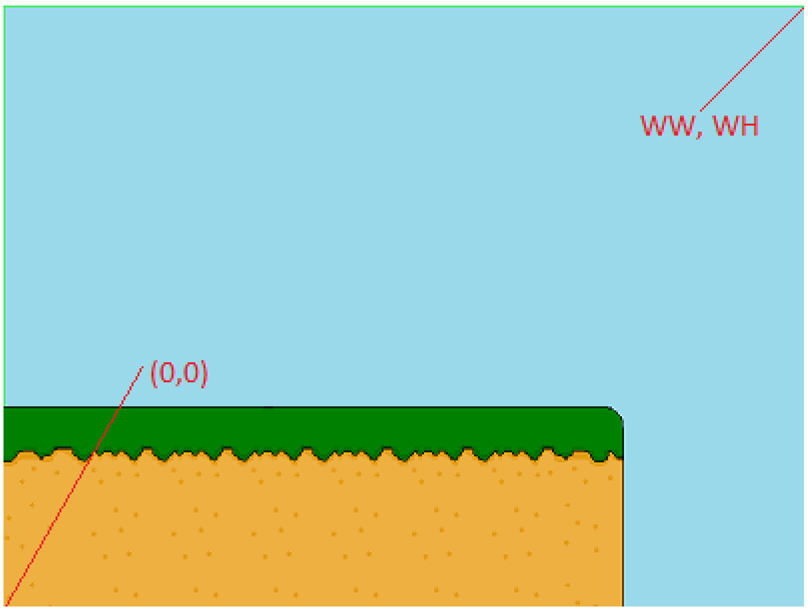


Figure 4.3:

4.3 File Format

Listing 2 Scenary File Format.

```
$SAVE\_TILESET()$
$SAVE\_BACKGROUND()$
$[int]internal\_window\_height$
$[int]internal\_window\_width$
$[double]gravity$
$[int]tile\_number\_y$
$[int]tile\_number\_x$
for $i = 0 \ldots Number\_tiles\_Y$ do
    for $i = 0 \ldots Number\_tiles\_X$ do
        $a \leftarrow 2$
    end for
end for
```

CHAPTER

5

SCENARYCOLLIDIBLEOBJECT

Is the most important class. It defines collisions with the scenary, with another objects, enemies etc.

5.1 Major Subclasses

- **PlayableCharacter** A class to define implementations of the game character.
- **Enemy** A class to define implementations of the enemies.
- **Item** A class to implement items used in the game.

5.2 Properties

boolean attack: check this object attack boxes? If this object can attack others, this must be true.

boolean receiveAttak: This object can receive attacks? if yes, this must be true.

boolean checkScenaryCollisionBoxes: This object collides with another scenary Collision Boxes?

boolean multipleCollisions:
int facingSide: Wich side is this objetct looking?
boolean render: Must be rendered? default=true;
boolean onFloor: Is this object touching the floor?
boolean onCeil: Is this object colliding some tile above it?
Vector2D position: The position in the scenary (not screen);
Action actualAction: The actual object action;
Scenary scenaryReference: The scenary this object is;

CHAPTER

6 GAMEITEM

The class GameItem describes several items in the game.

CHAPTER

7

ACTION

7.1 Introduction

The action class represents one movement that the character performs when one event happens. The player constrols the character by keyboard, mouse, etc. and a keyboard hit ou a mouse move can be caracterized as an event.

One action have spritesets, wich are activated in the *update()* method. The method updates the action state, character position and etc.

Each character implementation have a number of actions.

Only one Action can be active.

Each Action have a set of Spritesets. Each Spriteset have a name used to select it. It is stored in the **protected HashMap<String, Spriteset> spriteset** attribute.

Each Action have necessry keys, that referencies one Joypad Button.

Te activationkey is the key that activate the action: for example, the SPACE key makes the character jump; if the player press LEFT_ARROW or RIGHT_ARROW during the jump then the character will continue jumping but will move left. So, the SPACE is the principal key, called ACTIVATION_KEY.

Each subclass of Ation must implement the methods *update, canActivate*

& canDeactivate.

7.2 File Format

The action file is saved with its name. For example, to save the action "SIMPLE_WALK", the file will have be named "SIMPLE_WALK.act" and with the following format:

Listing 3 Action File Format.

```
[int]$CLASS\_NAME\_SIZE$
[byte*] $CLASS\_NAME$
[int] $SPRITESET\_NUMBER$
for Each Spriteset do
  [int]$SPRITESET\_NAME\_LENGHT$
  [byte*]$SPRITESET\_NAME$
  [int]$NUMBER\_OF\_LEFT\_SPRITES$
  for Each Left Sprite do
    [image] $Write\_Image()$
    [double]$DISPLAY\_TIME$
    [double] $ScenaryCollisionBox\_LowerLeftPoint\_x$
    [double] $ScenaryCollisionBox\_LowerLeftPoint\_y$
    [double] $ScenaryCollisionBox\_UpperRightPoint\_x$
    [double] $ScenaryCollisionBox\_UpperRightPoint\_y$
    [int] $NUMBER\_OF\_ATTACK\_BOXES$
    for Each ATK_BOX do
      [double] $ATK\_BOX\_LowerLeftPoint\_x$
      [double] $ATK\_BOX\_LowerLeftPoint\_y$
      [double] $ATK\_BOX\_UpperRightPoint\_x$
      [double] $ATK\_BOX\_UpperRightPoint\_y$
    end for
    [int] $NUMBER\_OF\_VULNERABLE\_BOXES$
    for Each ATK_BOX do
      [double] $VULNERABLE\_BOX\_LowerLeftPoint\_x$
      [double] $VULNERABLE\_BOX\_LowerLeftPoint\_y$
      [double] $VULNERABLE\_BOX\_UpperRightPoint\_x$
      [double] $VULNERABLE\_BOX\_UpperRightPoint\_y$
    end for
  end for
  [int]$NUMBER\_OF\_RIGHT\_SPRITES$
  ***SAME AS LEFT SPRITES***
end for
```

7.3 Class Members

This section show the necessary attributes in the class Action. These attributes are allocated in superclass Action but must be set in the subclasses.

String[] necessarySpritesets: The name of the necessary spritesets to this action can be used. Use the “addSpriteset” to add spritesets.

String[] spritesetsDescriptions: The descriotion of the necessary spritesets.

Hashtable<String, Boolean> toActivateRestrictions: The restrictions to activate this action. A hashtable containing “AnotherActionName, status(actve/inactive)”.

Hashtable<String, Boolean> toDeactivateRestrictions: The restrictions to deactivate this Action.

String name: The name of this Action.

boolean canDeactivate(): Return true if can deactivate, else return false.

boolean canActivate(): Return true if can activate the action.

Vector¡String¿ necessaryButtons: the button names that the action needs. This buttons must exist in the Joypad class.

void update(double timeElapsed): the most important method. Receives as parameter the elapsed time since the last update. Can use the joypad to verifies if some important key where pressed.

Vector¡String¿ activationButtonsNames: The name of the buttons that can activate the Action.

7.4 Creating Your Own Action

Each Action have a different behavior. Each one can have it’s own attributes. But all must:

- update it’s behaviour For example, for a jump, the action will make the character position increase or decrease. The action have acces to the joypad and its states and the key buffer. If its jumping and the player press LEFT the char will move left.

- Clear the Input making a call to *cleanInput()*.

The initialization **must**:

- Initialize *necessarySpritesets*: This is a String array and each position contains the name of a *spriteset* that must be created and set. Its used in the visual editor and used to verifie if the action is valid.

- initialize *spritesetsDescriptions*: This is a String array containing the description of the necessary spritesets.

- Set the Action's *name*: The name is essential, so, it must be initialized.

- Initialize *activationButtonsNames*: It is an array with the name of all buttons that can activate the Action. It is a String array with the Button names defined in *Joypad*. If the action need buttons to initialize (for example the action that runs when no key is pressed) the *activationButtonsNames = null*.

- Initialize *necessaryButtons*: It is an array of Strings, each position have the name of one Joypad button. Again, if the Action don't need an key the *necessaryButtons = null*.

CHAPTER

8

JOYPAD

8.1 Introduction

The Joypad class acts just like an joypad. It have some buton variables thar can be active or inactive. The character used this states to update.

All buttons have public access to simplify.

When a key is typed, the event handler (java.awt.event.KeyListener) verifies the key and store the status in "Hashtable<Integer, Integer> buttonStates". If the state is BUTTON_TYPED the action **must** set it to BUTTON_OFF in it's update method. Just use the "Action.cleanInput" method in the "Action.update" after doing the update.

The Joypad.buttonBuffer is a buffer with buttons in the order that are pressed (or threated).

8.2 Buttons

The buttons have 3 possible states:

- **public static final int BUTTON_OFF = 0**: the button is released.
- **public static final int BUTTON_TYPED = 1**: the button was typed (pressed one time and released).

- **public static final int BUTTON_HOLD = 2**: the button was pressed and hold.

If the button status = **BUTTON_TYPED** the character *update* method must set it to **BUTTON_OFF** after the use.

8.3 Attributes

Hashtable<String, Integer> buttonCodes: The name of buttons of controller, and the key code associated with it. The Action uses this map to update it state. For example: " 'BUTTON_UP',67", will map the button up to the key of asc CODE 67.

The Joypad can have as many buttons as the user need, but the button names must be accordling to the Action.necessaryButtons. For example, Action x have $necessaryButtons = [``BUTTON\_UP'', ``BUTTON\_X'']$ then the Joypad.buttonCodes must have the key maps for "BUTTON_UP" and "BUTTON_X".

8.4 Joypad Templates

Template creates a Joypad with some buttons.

Table 8.1: Possible collisions with the scenary.

Template	**Buttons**
TEMPLATE_JOYPAD_10_BUTTONS	UP, DOWN, LEFT, RIGHT, PAUSE, A, B, C, D, SPECIAL

CHAPTER

9 CHARACTER

9.1 Introduction

Each character extends the Character class. The *update* method must be implemented.

Each Character consists of a set of actions.

To create an character, must firstly create actions, then add the actions to character. After this, the method **String buildCharacter()** must be called in order to check and create character actions dependecy.

Each Action have its dependencies, in other words, to create an Action you need to create all Actions necessary to this Action. All these actions are added to the character.

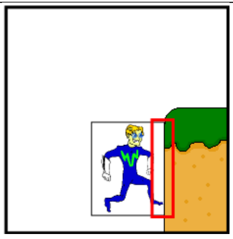
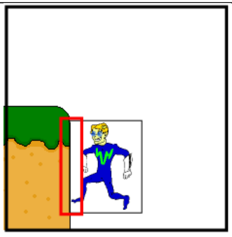
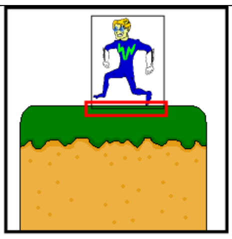
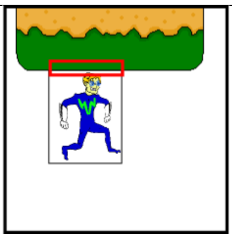
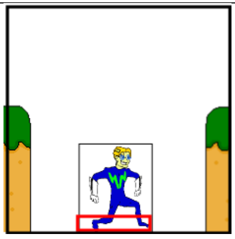
9.2 File Format

Listing 4 Character description File Format.

```
CREATE_DIR
[F][STRING] CLASS_NAME
[F][STRING]CHAR_NAME
for EACH ACTION do
  WRITE_ACTION_INSIDE_DIR
  [F][STRING]ACTION_NAME
end for
```

9.3 Collision Types

Table 9.1: Possible collisions with the scenary.

	COLLISION_SCENARY_LEFT_RIGHT
	COLLISION_SCENARY_RIGHT_LEFT
	COLLISION_SCENARY_UP_DOWN
	COLLISION_SCENARY_DOWN_UP
	COLLISION_SCENARY_OUT_SCENARY

9.4 The Update Method

The update method must be implemented in the subclass. It's the most important method, wich will determine all characters caracterstics.

Fisrtly, analise inputs in the Joypad.buttonBuffer. For each button pressed verifies if can activate the associated action (by the ACTIVATION_KEY).

CHAPTER

10 PHYSICS

10.1 Collision Box

Each collision box have four points: upper–left, upper–right, lower–left and lower–right. This points represents the translated points: the points are created accordling to Sprite coordinates, but the collision is computed using the translated points (computed from the Character position). In the Character collision detection the actual Sprite must have its collision boxes translated to the actual Character position.

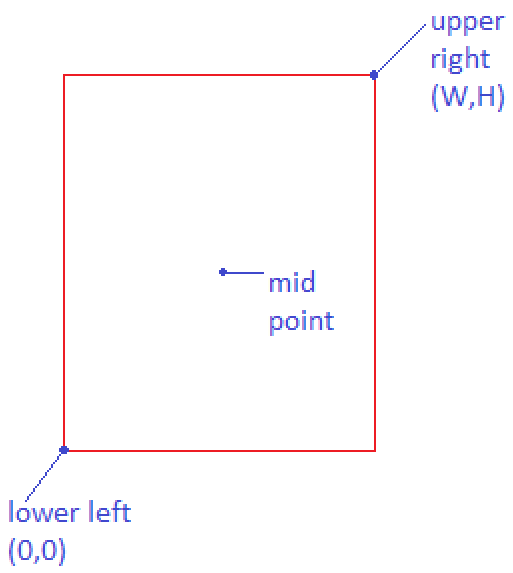


Figure 10.1: Collision Box coordinates Accordling to the Sprite coordinates.

10.2 Game Manager

Is a class to load the configuration

10.3 File Format

The game file is a text file containing:

Listing 5 Action File Format.

```
[int] INTERNAL_WINDOW_WIDTH
[int] INTERNAL_WINDOW_HEIGHT
[int] SCENARY_NUMBER
[string] CHARACTER_FILE_NAME
for EACH SCENARY do
  [string] SCENARY_NAME
end for
```