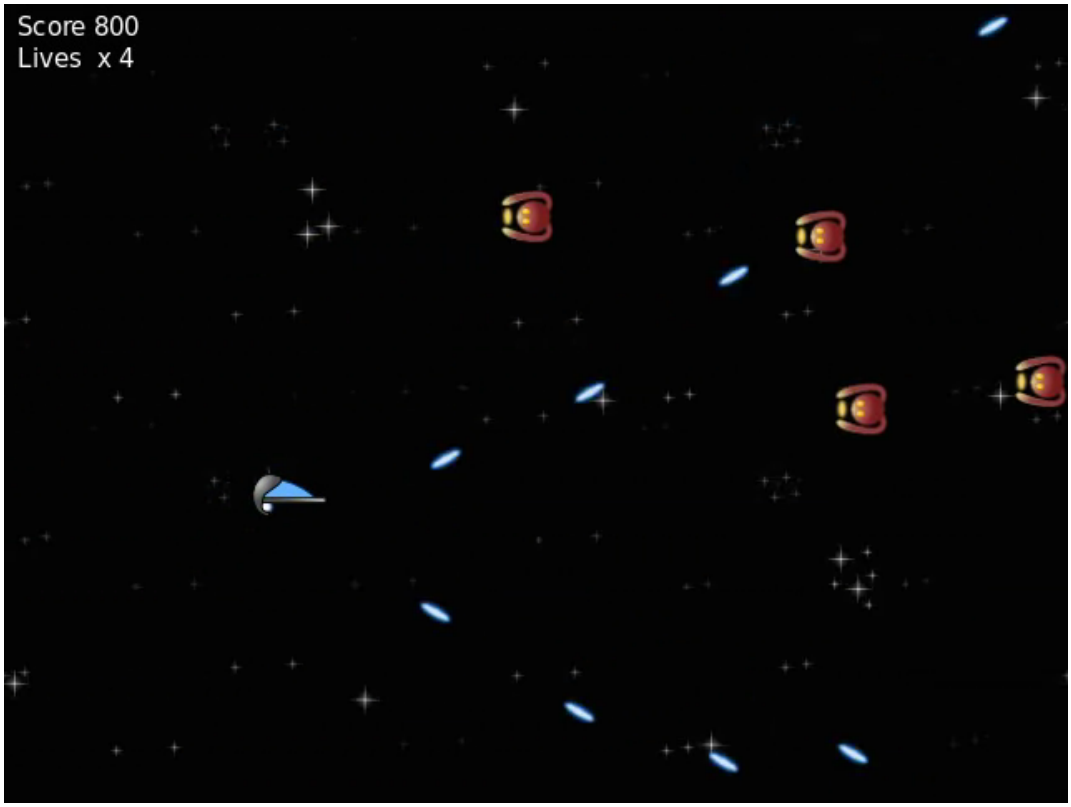
N-Six

An HTML5 video game



proposed by Maxime Daisy

<maxime.daisy@ensicaen.fr>

December 10, 2014

1 Introduction

1.1 Description

This project aims at programming a basic video game using HTML5 technologies. We propose to develop a shoot'em up game.

1.2 Required skills

- HTML5 Canvas
- Animation
- HTML5 Audio
- Basic collision detection

1.3 Organization and schedule

- Work in groups (5 students)
- A project manager
- One day

1.4 Deliverables

- Source code
- Oral presentation
- Demonstration

1.5 Provided material

The digital version of this subject, as well as the project asset package are available at the following address:
`https://daisy.users.greyc.fr/projects/ENSI3A_NSix/`

- graphics
  `graphics/`
- musics
  `audio/music/`
- FX sounds
  `audio/fx/`

Advice

Read *all* the subject carefully and make a chronological plan of the development.

2 Detailed description

2.1 Game process

The game starts when the web page has finished loading. The canvas and context are created and after the game is initialized, the game loop (infinite) is started and is called at a **60 frames per seconds** frame rate. To do so one can use either **window.setInterval** or **window.setTimeout**. The game loop is composed of two steps:

1. **Update phase**
   The goal of this phase is to update all game logic and objects. It consists of input management, object moving, collision detection etc.
2. **Render phase**
   This phase aim to draw all the objects of the game using their coordinates, colors, images etc.

Key words: javascript game loop, HTMLCanvasElement, HTMLCanvas2DContext

2.2 The Screens

Screen (also called "state") concept is present in (almost) all video games. It represents a state of the game, namely title screen for example. The game we develop contains the following screens:

1. **Load screen**
   Load screen appears at the beginning of the game, and simply display a load bar at the center of the game canvas until all the game assets are loaded: images and sounds (see *AssetManager* class). Then, the *title screen* appears.
2. **Title screen**
   This screen displays the title of the game, and a message like: "Press space bar to play". Once the user press the space bar, the choice screen appears.
3. **Choice screen**
   In this screen, the user choose between multiple spaceships.
4. **Level screen**
   In this screen, the core of the game takes place: game entities are created, score and lives are displayed, and the user controls the spaceship.
5. **Game Over screen**
   When no life remains to the player, this screen is shown. It displays the 10 best scores. When the user press the space bar, *title screen* is displayed.

Key words: state-based game

2.3 Game entities

2.3.1 Description

Shoot'em up games contains mainly four kinds of objects, namely:

1. **Player**
   An entity controlled by the user. It can be moved using arrow keys, and shoot **bullets**.

2. **Enemies**
   These entities try to prevent the player from finishing the game. They have (generally) predefined move patterns, and may also shoot **bullets**. In our basic game, enemies position is determined randomly at the beginning of the level.

3. **Bonuses**
   They can be of multiple utilities: give special powers or restore life for example. To start with, we propose to implement a unique bonus: *life bonus* (add a life to the player).

4. **Bullets**
   They are generally small objects launched by either the player or the enemies and can move in different directions and have different patterns. Once they are out of the screen, bullets are destroyed.

2.3.2 Management

At the beginning of the *level screen*, all entities are placed inside the scene at their initial position. During the game update, these entities are translated by the current level scroll speed. When they reach a specified x-coordinate ("entities activation line" in Fig. 1), they become *active*. In the latter state, entities (e.g. enemies, bonuses) follow their own move pattern. At "entities desactivation line", entities cannot act anymore, it prevents them to shoot bullets from out of the screen. Once their move pattern are finished, or they reach the "death line for level entities", entities are *destroyed* and must be removed from game object list.

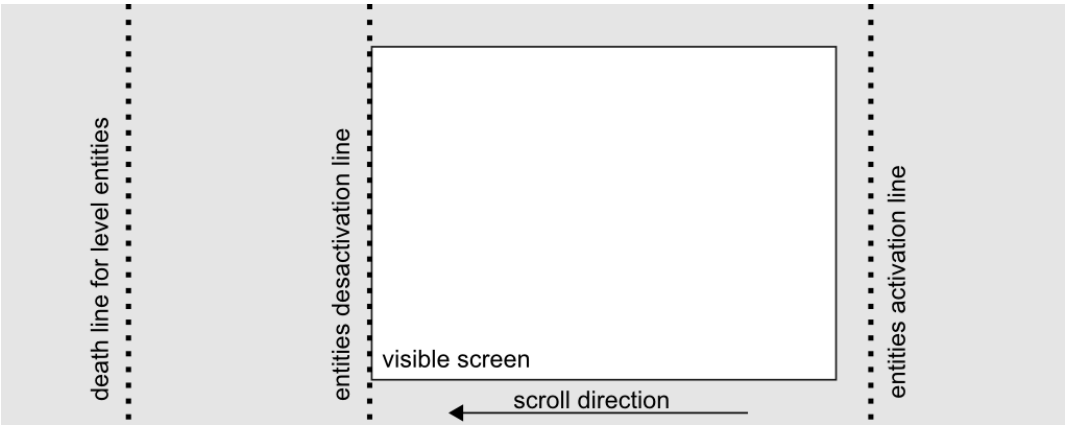


Figure 1: Game entities management illustration.

2.4 Input management

The **InputManager** object must be able to provide keyboard key states. Each time a key is pressed or released, key state array are updated. This module also specify the constants that correspond to the code of the keys used in the game.

Key words: javascript, key down, key up, key listener

2.5 Collision detection

2.5.1 Principle

In order to destroy the enemies, or the player when they are touched, or to know a bonus has just been catched, one must be able to detect whenever objects collide one another. This functionality is implemented using simple *rectangle* hitboxes[1].

2.5.2 Collision groups / collision filters

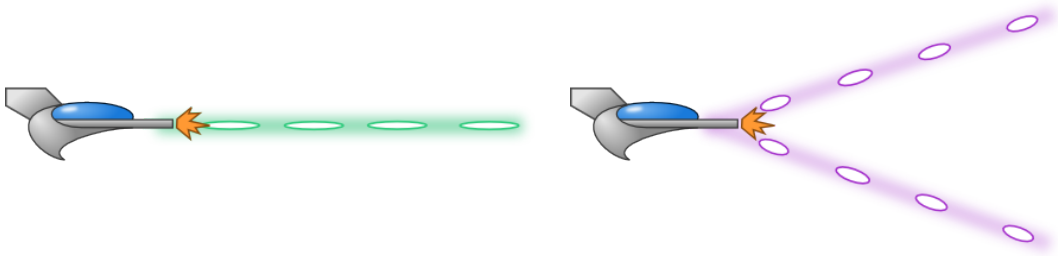
Every object is not able to collide with all other objects: player and its own bullets do not collide eachother for example. Collision groups and filter aim to solve this problem. If collision filters of an object A contain a certain group g, objects that belong to g cannot collide with A.

Key words: collision detection, hit box

2.6 Weapons

The player will be able to switch between *two weapons*:

1. **Straight weapon**
   This weapon is selected with the numpad key $[1]$ (see Fig. 2(a)).
2. **Two-sided Weapon**
   This weapon bullets, will be launched in two symmetric directions (see Fig. 2(b)). User select it using the numpad key $[2]$.



(a) Straight weapon.

(b) Two-sided Weapon.

Figure 2: Two different kinds of weapons.

When the user change its weapon, an FX sound is played (*audio/fx/WeaponChange.mp3*), and when the **SPACE** key is down, the player launch a continuous bullet burst.

2.7 Infinite background

In order to simulate a continuous displacement inside the space, *infinite background* are often used (see Fig. 3). The implementation of this graphical entity is quite simple: the elemental texture must be repeated all over the game canvas in a continuous way. The $offsetX$ and $offsetY$ values are the amount of move in the x and y axis respectively and must be updated at each frame.

Multiple infinite backgrounds can be used with different scroll speed (relative to a *virtual depth*) to create a parallax effect, this adds depth effect to the game.

[1] http://en.wikipedia.org/wiki/Hitbox

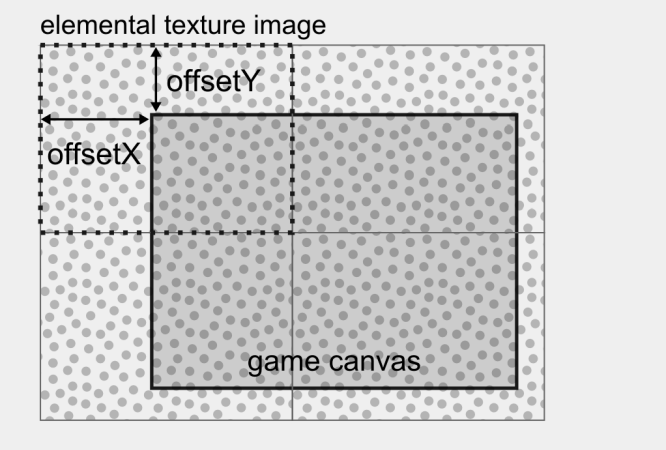


Figure 3: How infinite background works.

Key words: parallax effect, infinite background

2.8 Animations

In pixel-based 2D video games there are mainly two kinds of animations:

1. **Object-based animations**
   The principle is to draw and move multiple graphical objects to create the animation.
2. **Image-based animations**
   These animations are made of multiple images that are rendered successively.

In this project, we choose to implement the second type of animation.

Key words: time-based animation, frame-based animation, sprite-based animation

2.9 Audio management

In our game, Audio module will mainly be used for two things: 1) playing background music (often called BGM), and 2) playing sound effects (FX). While BGM are looping all along a level, FX are played only once when a specified event occurs (an enemy is destroyed, or player loose for example). After *Choice screen*, the BGM is faded out until *Level screen* appears.

Key words: HTMLAudioElement

2.10 Game System

- The player moves are bounded by the screen coordinates, meaning that it cannot go out of the screen.
- When the player is touched (enemy bullet, or enemy), the number of life decreas by one, an animation of explosion is played at its current position, and the players is replaced at its initial position. Also, a sound is played (*audio/fx/Loose.mp3*)and the player begins blinking for a certain time (let's say about 3 sec). During this moment, it becomes *invincible*.
- When an enemy is destroyed, the score player is increased by 100 points.

2.11 Saving the scores

Commonly, games are able to keep the best scores of the players. At the *GameOver screen*, we propose to save the player score (server side), load all scores (from the server) and display them on the screen.

Key words: ajax loading/saving, JSON, XML

3 Annexes

3.1 Diagrams

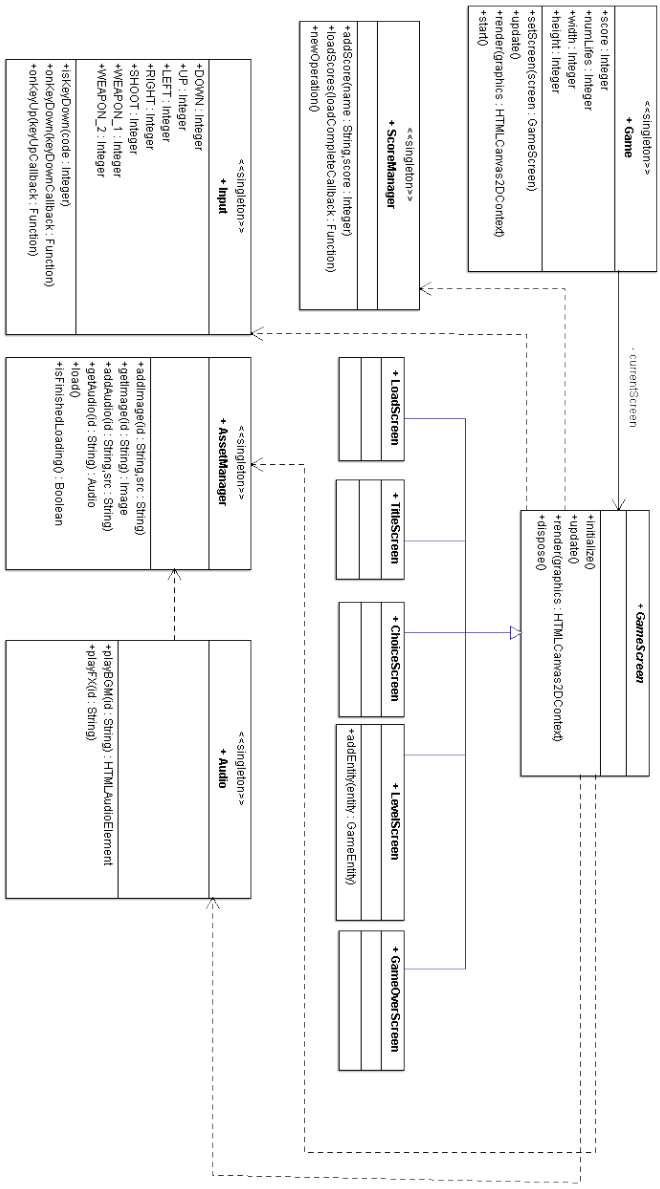


Figure 4: Core game class diagram.

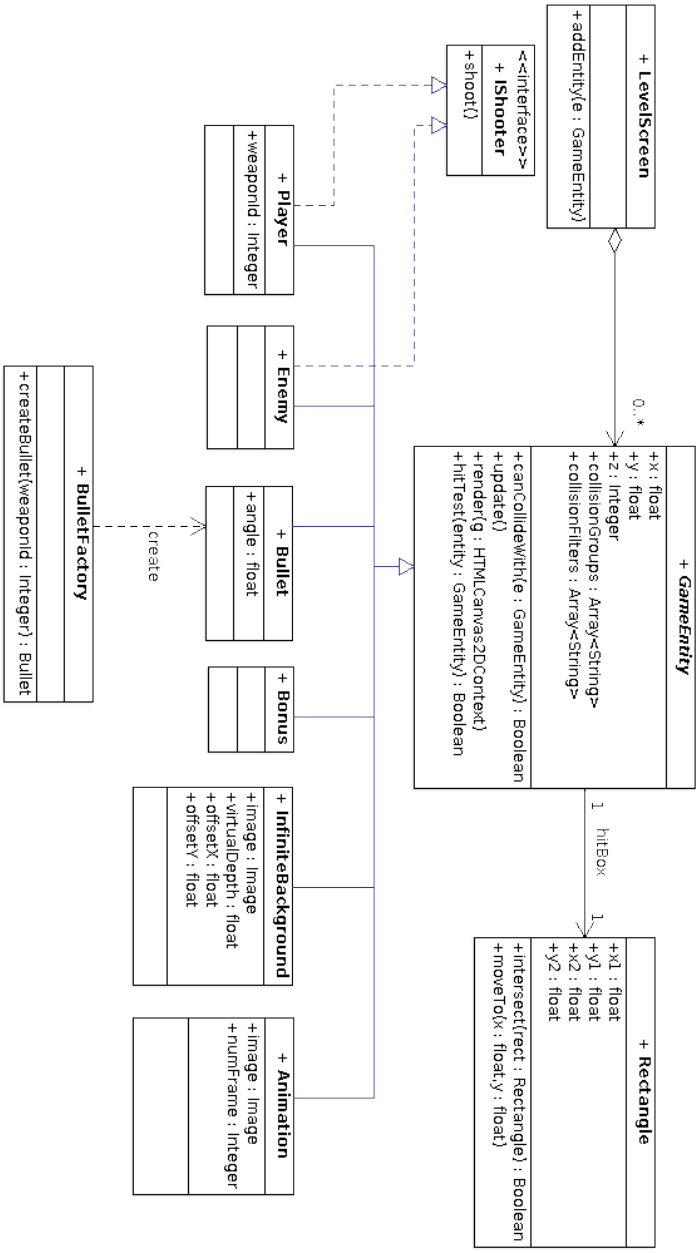


Figure 5: Level screen class diagram.

3.2 To go further

In this section we propose some additional content to put inside the game.

- Create a bonus (a bad bonus in fact) that reduce the field of view of the player.
- Implement a double-buffered graphics rendering.
- Add a key that allow the user to change spaceship move speed.
- Often, the user in shoot'em up games can launch huge bombs that destroy all the enemies on the screen...
- Create a module that loads the game levels from the server, and creates game entities dynamically by reading these levele.