# HTML 5 Arcade Game

Last updated September 11, 2015

by Louis Yang

# Overview

**Entry Points**

HTML Script
→ Frogger.init(canvas)

User Key Press
→ Frogger.keydown_handler(e)

Canvas repaint
→ Frogger.render(time)

**Object Hierarchy**

```
Frogger (global)
 |--MyCanvas _my_canvas
 |--CharSelection _char_selection
 |--Object _images
 |--Object _app_state
 |
 |--Game _game
     |--Map _map
     |
     |--Player _player
     |--Enemy[] _bugs, _cars, _fireballs
     |--Dragon[] _dragons
     |
     |--EnemiesController
     |--TempEffects _temp_effects
         |--CircularBuffer _text_effects_buf
         |--CircularBuffer _exp_effects_buf
```

**Inheritance**

```
GameObj
  |→ Player
  |→ Enemy
       |→ Dragon
```

**Running the Game**

The grayscale effect in the game is done in a method shown in the Udacity class, and this has security issues. As a result, the game must be run on a server to see the grayscale effects:

```
python -m http.server
localhost:8000
```

Alternatively, at the top of the "Frogger.js" file there is:

```
Frogger._enable_grayscale = true;
```

Changing this to "false" will disable the grayscale function, and allow the code to work by just double clicking on the index.html.

# Frogger

**Purpose**

This object / namespace contains global functions and variables for the whole game.

**HTML**

```
<canvas id="canvas" width="1100" height="650"></canvas>

<script type="text/javascript">

  var canvas = document.querySelector("#canvas");
  Frogger.init(canvas);
  window.addEventListener("keydown", Frogger.keydown_handler);

</script>
```

## init()

**Call Tree:**

```
Frogger.init()
  |
  |→_load_images()
  |     |→_load_image(...)
  |
  | (after images are done loading)
  |→_load_images_done()
        |→ CharSelection::render()
```

init() is triggered from the HTML. After all images have been loaded, the canvas is rendered as the character selection screen.

The "my_canvas" object is created in "init()" right away. Most objects have to wait until "load_images_done()" – because they need the "_images" object.

# _app_state

**Purpose and Initialization**

Application state information. Initialization happen inside "Frogger._load_images()".

**Object Members**

| | |
|---|---|
| string state | Name of the state |
| object param | The meaning of this variable depends on the state. |

**state**

"loading images" – you might see this screen if the needed images are not found
"character selection"
"first frame" – first frame of the game involve special treatment
"game running" – most of the time will be spent in this state
"level completed"
"game lost"

**Usage**

| state | param |
|---|---|
| "loading images" | num_images – number of images to load<br>total_images – total number of images |
| All Other States | null |

# _images

Frogger._images contain references to various images.

**Example:** "Frogger._images.boy" will get the image that is stored on the file " char-boy.png". The line of code that makes this connection is:

```
Frogger._load_image("char-boy.png", "boy");
```

# render(time)

**Purpose**

This function runs the animation loop of the game.

**Trigger**

Look for "render(null)" lines in the code. These are the first calls to the "render(time)" function. After the first call, the "render(time)" function schedules further calls via "requestAnimationFrame(Frogger.render)".

**time parameter**

When the browser calls "render(time)", the "time" is a timestamp in milliseconds.

**Frogger._old_time_stamp**

To prevent rendering too much, the old "time" stamp is stored. This mechanism can be used to reduce the framerate if needed.

**Frogger.render() Calls when starting a new game level**

```
Frogger.render(null);
  |→ requestAnimationFrame(Frogger.render);
...
Frogger.render(time); // time != null, state = "first frame"
  |→ requestAnimationFrame(Frogger.render);
  |→ Frogger._game.first_frame(time);

Frogger.render(time); // time != null, state = "game running"
  |→ requestAnimationFrame(Frogger.render);
  |→ Game::keydown_handler(Frogger._keydown_event, time)
  |→ Game::render(time)
```

# keydown_handler(e)

**Purpose**

This function handles keyboard events.

**"game running" Handler and Key Filtering**

```
Frogger.keydown_handler = function (e) {
    ...
    // "game running" state
  else if (Frogger._app_state.state === "game running") {
    Frogger._keydown_event = e; // delay keyboard input to render() frame
  }
```

So in the "game running" state, the "keydown_handler(e)" function records the key event to "Frogger._keydown_event". Eventually, the browse will call "render(time)" to redraw the canvas. At that point the keyboard input is processed. The "render(time)" function has the "time" timestamp. That timestamp is used to reduce the number of keydown events if the user is always holding down the same key.

```
Frogger.keydown_handler(e)
  | // "game running" state
  | Frogger._keydown_event = e; // delay keyboard input to render() frame
  |
  |--(inside the animation loop)
     Frogger.render(time)
        | // process keyboard input here
        |→ Game::keydown_handler(Frogger._keydown_event, time);
```

# MyCanvas

**Purpose**

Helper class for working with "canvas". Contains useful drawing methods.

**Creation**

There is only one MyCanvas object and it is created in "Frogger.init(...)".

**Interface**

| | |
|---|---|
| canvas | The HTML element "canvas" |
| context | 2d context from "canvas" |
| | |
| **Helper Functions** | |
| draw_text_centered(...) | Draw text that is centered on the canvas. |
| set_background(...) | Draw a rectangle that covers the whole canvas. |
| grayscale_filter(...) | Apply gray scale to a portion of the canvas. |

# CharSelection

**Purpose**

The CharSelection class represents the character selection screen.

**Creation and Initialization**

```
Frogger._load_images_done()
  |→ Frogger._char_selection = new CharSelection(...);
  |→ Frogger._char_selection.render();
```

**Keyboard Input Redirection to CharSelection:**

```
Frogger.keydown_handler = function (e) {
  if (Frogger._app_state.state === "character selection")
    Frogger._char_selection.keydown_handler(e);
  ...
};
```

# Game

**Purpose and Creation**

This object represents the game. There is only one Game object and it's created in "Frogger._load_images_done()".

**Code Leading to Game::init(...)**

Look for "_game.init(...)" in the code.

```
Frogger.keydown_handler(e)
  |→ CharSelection::keydown_handler(e)
  |
  |→ if (Frogger._char_selection.done)
        Game::init(..., level);
        Frogger.render(null);
```

level – this is where the level can be set

**Game::init() calls**

```
Game::init(...)
  |→ Map::init()
  |→ Player::config()
  |→ Player::init_new_level()
  |
  |→ EnemiesController::init(...)
  |     |→ _create_difficulty(...)
  |     |→ GameObj::spawn_between(...)
  |     |→ GameObj::set_location(...)
  |
  |→ TempEffects::set_text_styles(...)
```

# render(time)

**Call Tree**

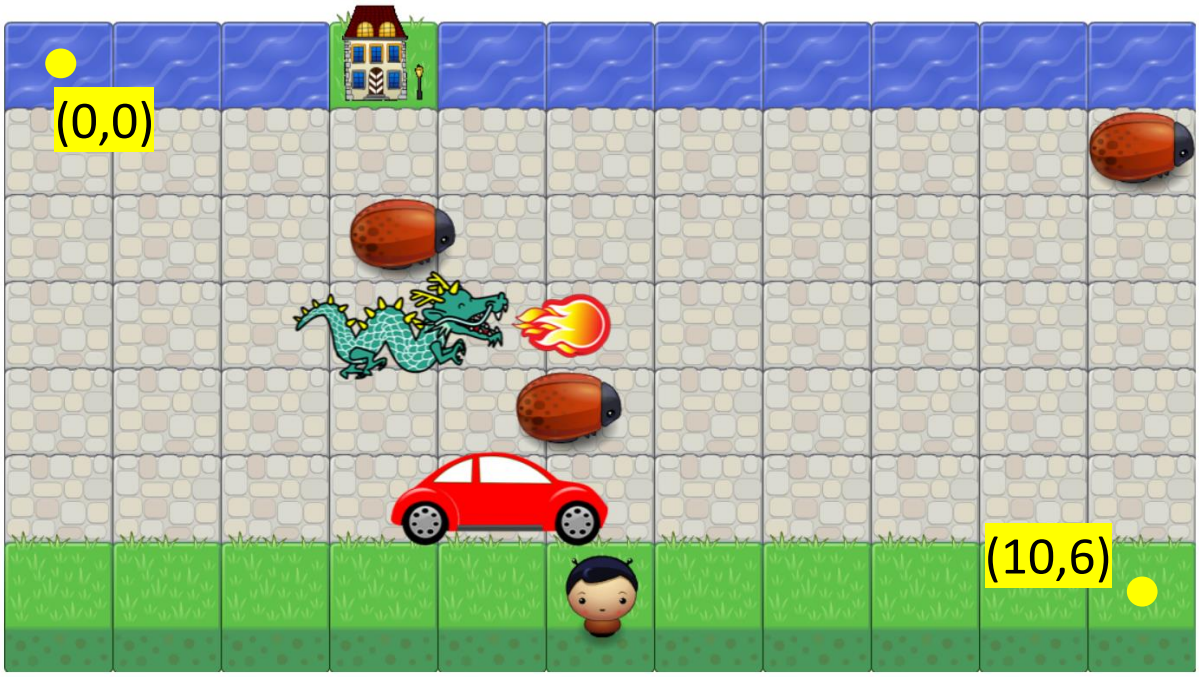
```
Game::render(time)
  |→ _update(time)
  |     |→ GameObj::update(time)
  |     |→ Dragon::update_fireball(...)
  |
  |→ _check_collisions(time)
  |     |→ Player::check_collision_bug(...)
  |     |→ Player::check_collision_car(...)
  |     |→ Player::check_collision_dragon(...)
  |     |→ Player::check_collision_fireball(...)
  |
  |→ EnemiesController::spawn(time)
  |
  |→ _draw_all(game_over_msg, game_over_color)
        |→ Map::draw()
        |→ GameObj::render()
```

# Map

- Provides map information. In particular **each square is 100 px wide and 80 px tall.**
- Draws the map.
- Converts logical coordinates to pixel coordinates.
- Can hit test a map coordinate to see if it's a water tile, or if it's the home tile.

**Logical Coordinates**

- x varies from 0 to 10
- y varies from 0 to 6




The home tile is chosen randomly for each game level.

# TempEffects

**Purpose**

This class is a group of temporary images and text that will be rendered on screen – kind of like GameObj. The difference is that each effect will be around for a limited time, and there are limited number of effects.

**Usage – Collision Events**

There is one "TempEffects" object in the game, created in Game(...).

The "Player::check_collision_xxx(...)" functions will add to this "TempEffects" object.

The "Game::_update(time)" function will call "TempEffects::update(time)" to disable old effects, which prevents them from being drawn.

The "Game::_draw_all()" will call the "TempEffects::render()" to draw temporary effects to the screen.

**Internal**

The TempEffects uses circular buffers. The number of concurrent effects on the screen will be limited. In the case of overflow, newly added effects will naturally overwrite oldest effects first.

# CircularBuffer

**Usage in TempEffects**

Although there is an "add(obj)" function, to reduce time spent on memory allocation / deallocation, the class can be used in such a way as to avoid adding new objects.

The "is_full()" function will return true once the circular buffer is full. Instead of using "add(obj)" to keep adding more objects, the code use "modify_oldest()" to retrieve the oldest object and modify its properties as needed.

# GameObj

**Purpose**

Represents a generic game object. Functionalities common to all game objects include movement and drawing. Data common to all game objects include logical coordinates, drawing coordinates, and a collision box.

**Usage**

When the user press a key to move the player, or when an enemy gets spawned, "**start_move(time, dx, dy)**" gets called.

During the animation loop, "**update(time)**" will update the object's position, and "**render()**" will draw the object.

Important call tree:

```
update(time)
  |→set_location(x, y)
```

**Derived Classes**

```
GameObj
  |→ Player
  |→ Enemy
       |→ Dragon
```

GameObj is not instantiated directly. Instead, one of the following is created:

- **Player** represents the character controlled by the user.
- **Enemy** can be a bug, a car, or a fireball.
- **Dragon** is a special enemy in that as it moves, it can spawn fireballs.

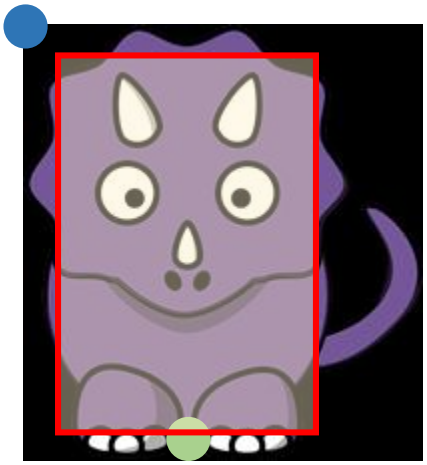
**_collided** member variable

Dragon, bugs, and cars only collide with the player once. After the first collision, the "_collided" flag is set to true, and further collisions are ignored. The "_collided" flag is reset to false when the game object is respawned.

# Coordinate System

Explanation of the many (x, y) coordinates used in the GameObj class.

See the "Map" section for a description of the logical coordinates (_x, _y).



The GameObj object has multiple "x" and "y" values, all for different purposes.

(_x, _y) – the logical coordinate on the game map, using 1 unit per game map square. This is shown as the green dot in the picture. This is the location of the object and is the source of all various coordinates.

(_x_draw, _y_draw) – the upper left hand coordinate in pixels, used to draw the picture. This is the blue dot in the picture.

collision box – the box used for collision detection; the red rectangle in the picture.

(_x1_coll, _y1_coll) and (_x2_coll, _y2_coll) – the upper left and lower right hand corners of the collision box. These coordinates are in pixels.

By default, the (_x, _y) coordinate refers to the center of the picture. But for the player characters, the (_x, _y) has been chosen to been center bottom of the picture. For these situation, the (_x_draw_offset, _y_draw_offset) describes the distances from the "center" of the picture to the upper left hand corner blue dot.

(_x1_col_offset, _y1_col_offset) – the distances from the center (green dot) to the upper left hand corner of the collision detection box. These distances are in pixels.

(_coll_box_width, _coll_box_height) – the size of the collision detection box.

**logical coordinate to pixel coordinate translation**

This is done via Map::to_x_pix(x), and Map::to_y_pix(y) functions.

# Player

**Creation and Initialization**

There is only one Player object and it's created in the "Game" constructor. There are two initialization functions:

- config(...) – called only before the very first level. This is a detailed configuration of the Player object.
- and init_new_level() – called before every level. This puts the player at the starting position and stops all motion.

# EnemiesController

**Purpose**

- init() – create enemies for the current map
- spawn() – spawn new enemies for the current map

**Difficulty Level Control**

```
EnemiesController::init(...)
  |→ _create_difficulty(level)
```

**Dragon and its Fireball**

Each dragon object is paired with a fireball object.

in Game::render()

```
  for (i = 0; i < this._dragons.length; i++)
    for (j = 0; j < this._dragons[i].length; j++)
      this._dragons[i][j].update_fireball(this._fireballs[i][i]);
```

So _dragon[i][j] is paired with _fireballs[i][j].

# spawn()

```
EnemiesController::spawn()
  |→_spawn_unit(...)
       |→_compute_respawn_distance(...)
```

**Algorithm**

No new enemy objects are created. Instead, when enemies move off screen, or have been eliminated, they are disabled and then eligible for respawning.

Spawning rule example: suppose map coordinate range is 0 ~ 10, and there are 2 bugs for a particular row. Bug A is on the left and bug B is on the right. Bug B will move off screen.

Since there are two bugs, then the average space between two bugs, ignoring the size of the bug for the moment, is 5. When bug B is removed from the screen, a bug to bug distance is chosen randomly, from the range of say 4 to 6 squares. Let say that a distance of 5.5 is chosen. When bug A's x coordinate is at least 2.5, activate bug B at -3. The large negative margin makes sure that larger objects, like the car, does not appear on the game map.

In the code there are further restrictions. For example, the size of the bug / car / dragon needs to be taken into account.

._bug_respawn_distances – this is where the 5.5 is stored.

**Spawning location difference**

The same function is used to spawn different enemies – but the starting size differ based on the size of the unit. The goal is to start and end the unit when it's totally off the map. So a larger unit, like the dragon, needs to start farther away from the map.

Bug – width = 1, start at -1, end at map size +1

Car and dragon – width = 2, start at -1.5, end at +1.5

# Coding Reference

## Frogger

| | |
|---|---|
| keydown_handler(e)<br>_load_images(),_load_images_done()<br>_random_int(min, max) | **_game, _app_state,**<br>_my_canvas, _char_selection,<br>_images<br>_keydown_event, _prev_keydown_code,<br>_prev_keydown_time |
| _init_level, _last_level, _enable_grayscale | |

### _images

car, boy, girl, dino, dragon, bug, explosion1, explosion2, explosion3, fireball, gem_blue, gem_green, gem_orange, grass, house, selector, star, stone, ufo, water

## MyCanvas

| | |
|---|---|
| Variables: | canvas, context |
| Functions: | draw_text(font, fill, stroke, text, x, y)<br>draw_text_centered(font, fill, stroke, text, y)<br><br>grayscale_filter(rect_list, change_alpha)<br>// "rect_list" - an array of rectangles, each rectangle item is<br>// encoded as (x, y, width, height)<br>// "change_alpha" - "true" allows change of alpha as part of the filtering<br><br>set_background(fill, stroke) |

## CharSelection

| | |
|---|---|
| Variables: | _my_canvas, _images<br>current_selection |
| Functions: | render(), reset(), get_player()<br>keydown_handler(e) |

## Game

| | |
|---|---|
| init(), render(), first_frame()<br>keydown_handler(e, time) | _my_canvas, _images, _map_settings<br>_home_square<br>_temp_effects |
| _player, _enemies_controller, _bugs, _cars, _dragons, _fireballs<br>_level, _win, _lost | |

## Map

| | |
|---|---|
| Map(my_canvas, images) | _num_col, _num_stone_rows, _square_width, _square_height, _header_space<br>_home_square |
| to_x_pix(x), to_y_pix(y) | render() |

# GameObj

| | |
|---|---|
| GameObj(my_canvas, image, map_settings) | _my_canvas, _image, _map_settings |
| set_location(x, y)<br>start_move(time, dx, dy)<br>render() | _x, _y, _x_draw, _y_draw, _x_draw_offset , _y_draw_offset |
| _enable, _is_moving<br>set_enable(enable) | _x1_coll, _y1_coll, _x2_coll, _y2_coll, _x1_coll_offset, _y1_coll_offset, _coll_box_width, _coll_box_height<br>_collided<br>set_collided(true) |
| spawn_between(x1, x2, y) | _speed, _dx, _dy, _time, _dest_x, _dest_y |

## Player

| | |
|---|---|
| _images, _x_start, _x_max, _y_max<br><br>_eats_bugs, _flyer, _breaks_cars, _hp, _hp_max | keydown_handler(e, time)<br>config(player_selection)<br>init_new_level() |
| check_collision_bug(bug, time)<br>check_collision_car(bug, time)<br>check_collision_dragon(bug, time)<br>check_collision_fireball(bug, time) | |

## Enemy

| | |
|---|---|
| Enemy(my_canvas, image, map_settings, speed) | start_move_to_right() |

## Dragon

| | |
|---|---|
| Dragon(my_canvas, image, map_settings, speed) | _fireballs_left, _fireball_x<br>reset_fireball(), update_fireball(fireball, time) |

# EnemiesController

| | |
|---|---|
| **init(level, my_canvas, images, map_settings)**<br><br>returns [bugs, cars, dragons, fireballs] | _level, _max_x, _square_width<br>_bugs, _cars, _dragons, _fireballs |
| | _bug_respawn_distances<br>_car_respawn_distances<br>_dragon_respawn_distances<br>**spawn()** |

## difficulty

ground_enemies, max_ground_enemies, ground_speed,

air_enemies, max_air_enemies, air_speed

# TempEffects

| | |
|---|---|
| TempEffects(...)<br>set_text_styles(player) | add_explosion(time, x, y),<br>add_text(time, text, x, y) |
| _text_effects_buf, _exp_effects_buf<br>_my_canvas, _explode_images<br>_stroke, _fill, _font | clear(), update(time), render() |

xxx_buf object properties: x, y, content, time, enable

## CircularBuffer

| | |
|---|---|
| CircularBuffer(max_size) | length(), is_full(), add(obj), get(index),<br>modify_oldest() |