Synchronous Retrogames in HTML5

Retrogames were played in the dark arcade rooms of the past millenium. Such games were written in assemby as to squeeze every bit of performance of hardware that wouldn't go past a few Megahertz. Nowadays, a mere smartphone harnesses several Megabytes of RAM and a multiple processor cores clocked at several Gigahertz each. Such power allows us to revisit game programming: ease of development takes the lead while performance is sidelined.

Web browsers are ubiquitous - and are quickly becoming the defacto platform to run multiplatform apps. Furthermore, the new web standard HTML5 greatly simplifies the deployment of dynamic application through the Javascript language. Notably, the canvas interface allows to draw objects, animate them, and intercept inputs from mouse or keyboards in a Web page.

The canvas API imposes a reactive programming paradigm: in order to update the canvas, one should register a callback function through `requestAnimationFrame()`. One has to do the same to process input.

The reactive limitations imposed by the canvas API is similar to what is used in control-command systems. Several languages, based on the synchronous dataflow paradigm, have been created in order to ease development of such systems. A program written in such a language processes a flow of events - player inputs - and produces a flow of actions corresponding to instructions that will be transmitted to the actuators.

In this internship, we aim at applying this programming model to the implementation of gameplay code. We will focus on the application of the synchronous dataflow formalism to retrogame programming:

- We describe a first approach implementing a Snake clone in pure javascript.
- We design a synchronous dataflow language integrating the HTML5 canvas API.
- We devise a compiler compiling from this language to Javascript. The compiler is written in Ocaml, and will be bootstraped through `js_of_ocaml`.
- We validate this second approach by implementing an other retrogame - this time using this brand new language.

Pierre: you should update all the following with the example we discussed today.

Consider the following javascript code, which simply moves a square from the left to the right:

```
var right_press = false;

document.addEventListener("keydown", keyDownHandler, false);
document.addEventListener("keyup", keyUpHandler, false);

function keyDownHandler(e) {
    if (e.keyCode == 39) {
        right_press = true;
    }
}

function keyUpHandler(e) {
    if (e.keyCode == 39) {
        right_press = false;
    }
}

function move() {
    if (right_press) {
        x += 2;
    }
    requestAnimationFrame(move);
}
```

The callback registration is cumbersome. Furthermore, one has to use many global variables, which lead to confusing code.

Now compare it to the much nicer *sdf* code:

```
node right = (right: int) with
    let right = keyboard_input 39;
tel;

node x (right: bool) = (x: int) with
    let x = if r then pre x else 0 -> pre x + 2;
tel;
```

Everything related to event programming is abstracted in the structure of the language.

Such code is then compiled to the following javascript:

```
function x() = {
    this.b_1 = true;
    this.x_2 = 0;
}

x.prototype.reset = function () {
    this.b_1 = true;
    this.x_2 = 0;
}

x.prototype.step = function (right) {
    var x = 0;
    var b = false;
    var x_3 = 0;
    var b = this.b_1;
    if (b) {x_3 = 0;} else {x_3 = x_2 + 2}
    if (right) {o = x_2;} else {o = x_3;}
    x_2 = 0;
    return o;
}
```