

Synchronous Retrogames in HTML5

Retrogames were played in arcades of the past millenia. Such games were written in assembly as to squeeze every bit of performance of hardware that wouldn't go past a few Megahertz. Nowadays, a mere smartphone harnesses several Megaoctets of RAM and a multi-core processor exceeding a frequency of one Gigahertz. 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 impose to use reactive programming: 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. In order to ease development in such systems have been created specialized languages based on Synchronous Data Flow theory. A program written in such a language is considered to process a data flow - player inputs - and produce a data flow corresponding to instructions that will be transmitted to actors.

A sharp mind may then get the intuition that a synchronous data flow language might be an efficient way to devise gameplay code.

In this paper, we will focus on the application of the synchronous data flow formalism to retrogame programming:

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

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;  
}
```