Coda Lisa: Collaborative Art in the Browser

Felienne Hermans
Delft University of Technology
f.f.j.hermans@tudelft.nl

Rico Huijbers
Amazon
huijbers@amazon.com

Abstract

This paper introduces Code Lisa: a collaborative programming environment in the browser that allows users to program one cell in a grid of cells. The cells can react to each other, but also to the environment, represented by sensor values on an EV3 Lego Mindstorms robot. By programming reactions to each other and to the sensors, a group of Coda Lisa users can together create a living, interactive art work. Users program the cells using JavaScript, and can experiment locally with their cell in a live editor created for this purpose, which shows a preview of the cell's behaviour. Once ready, the cell's program can be submitted to the server so it can be rendered in the shared grid of cells, by a second Coda Lisa client application: the viewer. In addition to the implementation of Coda Lisa, this paper also describes several plans for future development.

1. Introduction

In recent years, live programming has attracted the attention of many, in all sorts of different domains, from education [1] to music programming [4]. The liveness of programming languages and environments enables quick feedback, which in turn powers creativity. Modern web browsers make it easier than ever to support live coding, because they are sophisticated platforms for rendering and scripting.

Programming environments which enable users to create drawings or more elaborate artwork are common, starting in the eighties with the Logo programming language to modern day with sophisticated tools like Processing [5].

In this paper we propose Coda Lisa, a program that enables making art through collaborative live programming. Coda Lisa is a live coding environment that enables users to collaboratively create a living artwork. Code Lisa users control a single *cell* in a grid, where other cells are drawn by other Coda Lisa users. By reacting to other cells, as well as to the physical environment around the players by reading sensor values, cells can be programmed to behave interactively, and together form one artwork as shown in the Figure below.

This paper describes the implementation of Coda Lisa and proposes a number of future extensions to be developed.

2. Implementation

2.1. Editors

The central component of Coda Lisa is a canvas divided into a grid of cells, whose contents are controlled by the users. This is the view shown in Figure 1.

Coda Lisa users control their cell by implementing two JavaScript functions: an `init` function and a `draw` function. The `draw` function can be declared as taking 1 to 3 arguments which will be explained below.

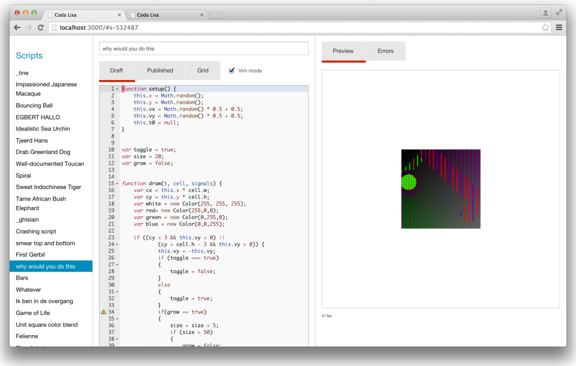
In the most basic form, `draw` takes a `cell` argument that represents the cell that an agent program is about to draw into. Users write code that will set color values for the pixels in the cell. For example, a red circle can be created with this piece of code:

```
function draw(cell) {
    for (var y = 0; y < cell.h; y++) {
        for (var x = 0; x < cell.w; x++) {
            var color;

            if (dist(x, y, 50, 50) < 10)
                color = new Color(255, 0, 0);
            else
                color = new Color(255, 255, 255);

            cell.set(x, y, color);
        }
    }
}
```

When run locally, in a user's browser, the code runs in the central cell of a grid of 9 cells, as shown by the picture below:



```
function setup() {
    this.x = Math.random();
    this.y = Math.random();
    this.vx = Math.random() * 0.5 + 0.5;
    this.vy = Math.random() * 0.5 + 0.5;
    this.t0 = null;
}


var toggle = true;
var size = 20;
var grow = false;


function draw(t, cell, signals) {
    var cx = this.x * cell.w;
    var cy = this.y * cell.h;
    var white = new Color(255, 255, 255);
    var red= new Color(255,0,0);
    var green = new Color(0,255,0);
    var blue = new Color(0,0,255);

    if ((cy < 3 && this.vy < 0) ||
            (cy > cell.h - 3 && this.vy > 0)) {
            this.vy = -this.vy;
            if (toggle === true)
            {
                toggle = false;
            }
            else
            {
                toggle = true;
            }
            if(grow == true)
            {
                size = size + 5;
                if (size > 50)
                {
                    grow = false;
```

Figure 1: *The Coda Lisa editor: left is a list of all available programs, in the middle the JavaScript code can be entered, right, a preview of the cells behavior is shown*

The above interface is a Coda Lisa client application called the *editor* and is live: On every code change, the functions are compiled, and when compilation succeeds the `init` and draw functions on the data object are replaced, while the draw function keeps being executed at 30Hz.

To allow the user to have persistent data between live code reloads, we store data on a long-lived object that is outside the user code, and we call `init` as few times as possible. We use the prototypal facilities of Javascript to move the functions onto the object after they have been compiled.

`init` is in executed in the context of the data-holding object the first time the agent is run, and again on-demand when the user wants to change the `init` code. The context allows the user code to use `this.` to access long-lived variables.

When a user is satisfied with his cell's programming, she can submit the code to the Coda Lisa server. The execution of the user programs is then run by a second client application called the *viewer*, which executes all programs together on a big screen. This is when the user's program can start to interact with the cells created by other Coda Lisa users. When a user submits the code, the viewer calls the `init` method which is run once, and then calls the `draw` function at 30 Hz. The `draw` function has a *time* parameter that users can read (where time is given in fractional seconds), which allows users to create animated cells. For example, by changing the code of the previous example slightly, we create a red circle that moves left at a rate of 10 pixels per second:

```
function draw(cell, t) {
    for (...) {
        var x0 = modulo(50 - t * 10, cell.w);
        if (dist(x, y, x0, 50) < 10)
            color = new Color(255, 0, 0);

        // ...
    }
}
```

2.2. The environment

In addition to the simple variant of the draw function in which just time is used as a parameter, users can also have their cell react to the environment. The first environmental variable is the color values of the canvas in the last iteration. The `cell` object that is passed to the `draw` function has a `get` function, which will return the color of any pixel, relative to the cell's own coordinate system. By mixing past and future colors in the cell's own region, the program can create *blending* effects.

An agent cannot only write but also read pixels outside its own cell's boundaries, but an agent is not allowed to write outside its own cell. This reading outside the cell gives a user the opportunity to respond to the pixels in neighbouring cells. She can, for instance, create a *reflection* cell that mirrors a neighboring cell. To test their programs in this manner in the 3x3 grid in their local editor, the cells surrounding cells by default come pre-loaded with an agent program that responds to mouse events by painting pixels.

Finally, users can incorporate sensors values from the real world into their programs. For this, we have built a Lego EV3 Mindstorms robot with touch, color, sound and distance sensors. Using a simple request-response protocol over a serial connection to the EV3 brick, the sensor values are exposed to Coda Lisa programs in a third argument of the `draw` function.

For example, the following program draws a circle whose radius depends on the value of the distance sensors, with smooth transitions by blending the pixels of the previous frame:

```
function draw(cell, t, sensors) {
    for (...) {
        var color;

        if (dist(x, y, 50, 50) < sensors.dist * scale)
            color = new Color(255, 0, 0);
        else
            color = new Color(255, 255, 255);

        color = color.mix(cell.get(x, y), 0.7);

        cell.set(x, y, color);
    }
}
```

3. Related work

We took our inspiration for Coda Lisa from various existing systems. We for example looked at Processing, a programming language originally developed as a tool to teach kids to program, which evolved into a tool for artists and researchers [5]. Similar environments are Gibber [2], and Lich.js [3], both browser based live coding language for music and graphics.

Coda Lisa shares the idea of being a programming environment for art with these systems, but, while they do support some form of collaboration, they are not solely meant for it, like Coda Lisa is.

4. Conclusions & Future extensions

This paper describes a live programming environment in the browser, which lets users live program agents to control cells in a grid. The cells can vary depending on time and their previous color values, but can also react to colours of adjacent cells, or EV3 sensor values. In this way, Coda Lisa users collaboratively program a living, interactive artwork.

This paper presents a first version of Coda Lisa, which to date only has been demonstrated and used at one event. We foresee the following future extensions.

4.1. More environmental variables

A first extension would be to add more environmental variables in the form of *live data streams*, like stock tickers, weather data or seismographic information.

Furthermore, we could allow people to connect their mobile devices to the shared canvas, not only to act as an additional viewer, but also to send the devices sensor values, like gyroscopic sensors, to Coda Lisa, or to use the devices camera as input for a certain cell.

Finally, it would be great to extent Coda Lisa such that it can be played on a device with touch capabilities and have cells react to the screen being touched.

4.2. Better debugging and more directness in the editor

When users programmed their cells in the first Coda Lisa workshop, we noticed the need for more support in understanding the behaviour of their cells. What would greatly help here are features like the possibility for a user to *rewind* their animation and replay it slowly. When we ran the workshop, we noticed users mimicking this type of debugging by inserting `sleep` commands in their code, so this is something Coda Lisa should support. To make the replay more insightful, we could even highlight the current line of code being executed to improve understandability and make Coda Lisa more suitable for educational purposes.

4.3. A shared canvas

Finally, we would like to explore the possibility to, instead of dividing the canvas into cells, have all agents share the canvas. This increases opportunity for aesthetically pleasing compositions, as the grid structure leads to sharp edges in the final composition. However, this will also increases the complexity of the programming, both on the side of the user and on the Coda Lisa side.

References

[1] Code monster home page. http://www.crunchzilla.com/code-monster. Accessed: 2015-05-10.

[2] Gibber home page. http://gibber.mat.ucsb.edu/. Accessed: 2015-05-10.

[3] Lich home page. https://www.chromeexperiments.com/experiment/lichjs. Accessed: 2015-05-10.

[4] Pam Burnard, Nick Brown, Franziska Florack, Louis Major, Zsolt Lavicza, and Alan Blackwell. *Processing: A Programming Handbook for Visual Designers*. http://static1.squarespace.com/static/5433e132e4b0bc91614894be/t/5465e778e4b02ea3469103b0/1415964536482/research_report_dc_02.pdf, 2015.

[5] Casey Reas and Ben Fry. *Processing: A Programming Handbook for Visual Designers*. MIT Press, 2014.