UN LENGUAJE DE PROGRAMACIÓN PARA EL CUERPO

ABORDAJE RETÓRICO SOBRE LA PROGRAMACIÓN Y SU ARTICULACIÓN CON LAS ARTES PERFORMÁTICAS.

Director: Emiliano Causa
Autores: Francisco Alvarez Lojo, Ezequiel Rivero, Ignacio Siri, Agustin Bacigalup, Daniel Loaiza, Hernán González Moreno, Carolina Ojcius. (integrantes del Laboratorio EmmeLab)

INTRODUCCIÓN

El emmeLab es un Laboratorio de investigación y experimentación en nuevas interfaces para el arte, dependiente de la Secretaría de Ciencia y Técnica de la Facultad de Bellas Artes, UNLP. Durante el año 2014 se planteó generar un sistema de proyecciones programable que sea controlado a través de la captura de movimiento en tiempo real. Un intento de generar un entorno que permita una suerte de LiveCoding combinado programación en vivo con Danza Contemporánea o Expresión Corporal.

En primera instancia fue necesario generar un tipo de lenguaje de programación pensado específicamente para ser controlado mediante captura de movimiento. Se llevó a cabo una investigación respecto a los elementos que componen o definen un lenguaje de programación, se plantearon diversos conceptos sobre las partes de un lenguaje y sobre paradigmas de programación reconocibles.

Se determino que el resultado del lenguaje sea específicamente visual, la salida de proyección estaría dada por un sistema de partículas, basados en que este sistema articula muy bien con las artes escénicas, además que el emmeLab tiene mucha experiencia en el desarrollo de entornos generativos visuales.

Por otro lado se consideró muy adecuado un sistema de partículas dado que su comportamiento está fuertemente ligado al movimiento y puede generar fácilmente una conexión semántica con la danza o expresión corporal donde el movimiento y el cuerpo son los protagonistas.

Kinect
mocap
motion capture
interactivo
detección de movimiento
captura de movimiento
artes performáticas
danza contemporánea
cuerpo
programación esotérica
sistemas de partículas
arte generativo
generatividad

Respecto a la captura de movimiento se utilizó el sistema Kinect para tomar y pre-procesar la información, definimos una serie de gestos, poses o caracteres del movimiento que serían detectadas como "entradas" para intervenir el "código".

Tras una breve investigación en danza se advirtió que existen ciertos parámetros "técnicos" en la disciplina que son utilizados para describir un paso o movimiento, se intentó explotar estos parámetros y, al mismo tiempo, que los gestos propios del lenguaje no comprometieran la posibilidad de una interpretación estética o natural por parte del performer.

Finalmente se genero un sencillo sistema de partículas ampliamente modificable, aunque en su primera instancia de desarrollo no ofrece gran variabilidad. Sin embargo la estructura del sistema está diseñada para poder incrementar fácilmente y de manera colaborativa las posibilidades que ofrece.

El sistema es controlado a través de mensajes OSC, estos mensajes pueden enviarse entre diferentes aplicaciones, de modo que el control del sistema puede darse por cualquier medio, más allá que la intención principal apunta a que sea utilizada con captura de movimiento.

LIVE-CODING

El live-coding es una disciplina que consiste en transformar el acto de programar en un acto performático. El programador-performer se presenta ante el público y escribe con su herramienta predilecta un programa que genera visualizaciones y/o musicalizaciones en tiempo real. La pagina TOPLAP[1] es el principal referente de live-coding y en ella existe una suerte de manifiesto que fundamenta el live-coding. Allí se mencionan ciertas "buenas costumbres", como por ejemplo que es mejor aceptado utilizar herramientas propias, pero un código debe generarse desde cero, y la fuente debe ser visible (incluso cuando el producto del programa es también visual). [2 3 4]

Figura 1 a.

Sesiones de Live Coding

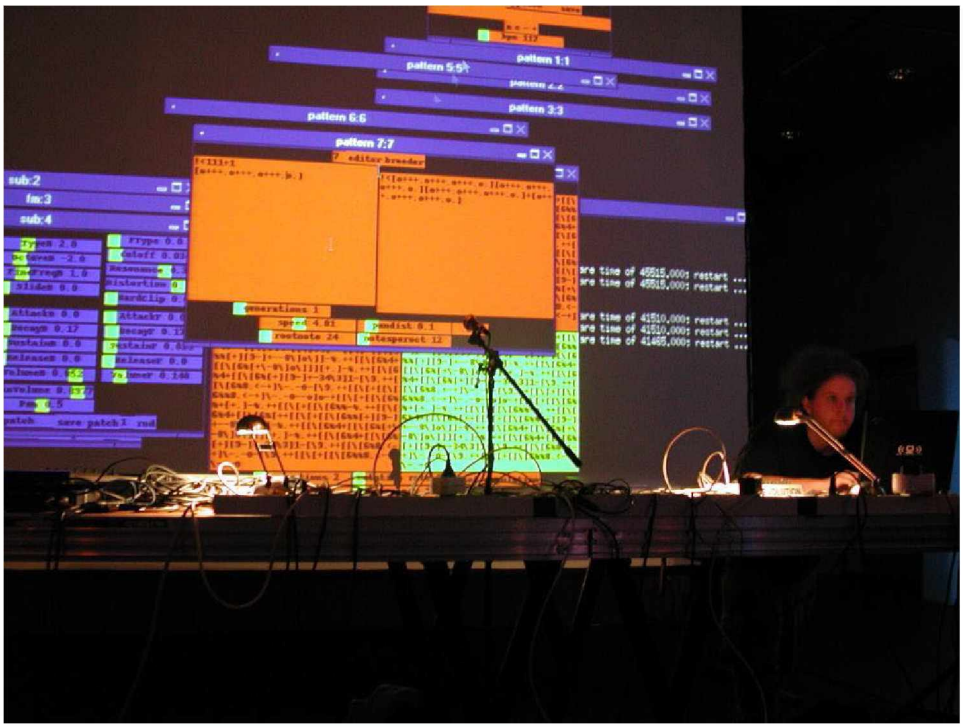


Figura 1 b.

Sesiones de Live Coding

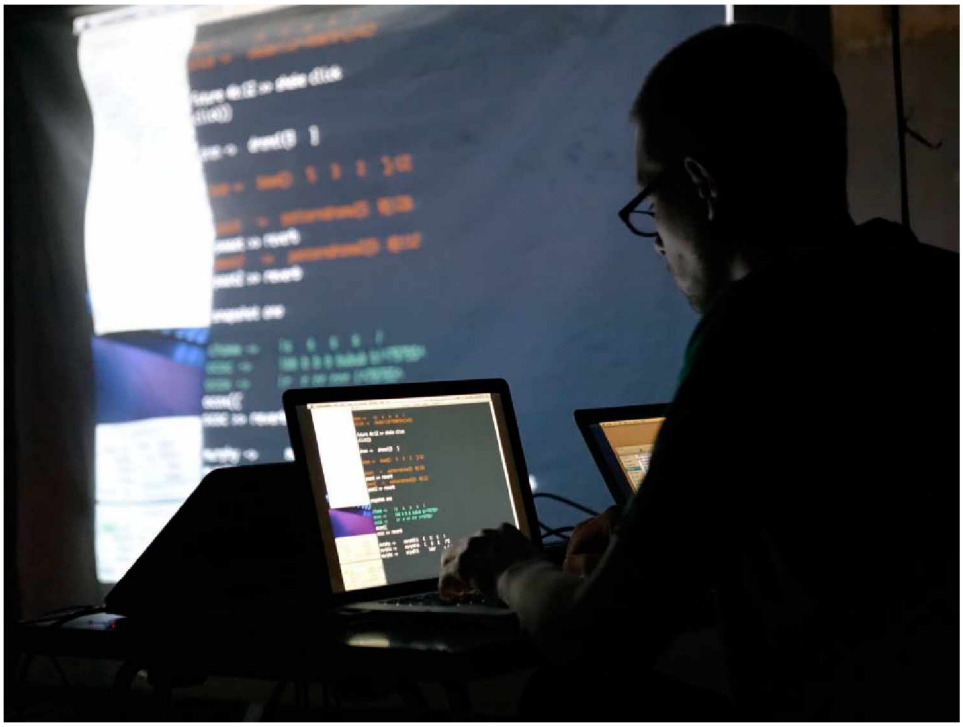


Figura 1 c.

Sesiones de Live Coding

Aunque el proyecto surge en parte basado en el live-coding, no consideramos que se pueda clasificar puramente dentro de esa disciplina, pues se busca generar una expansión de la danza hacia la tecnología y la programación, y no desde el live-coding hacia la danza. Los lineamientos del desarrollo se encuentran condicionados por la intención de no restringir al bailarín/performer lo cual no da lugar a aumentar las limitaciones en un proyecto tan experimental.

LENGUAJES DE PROGRAMACIÓN

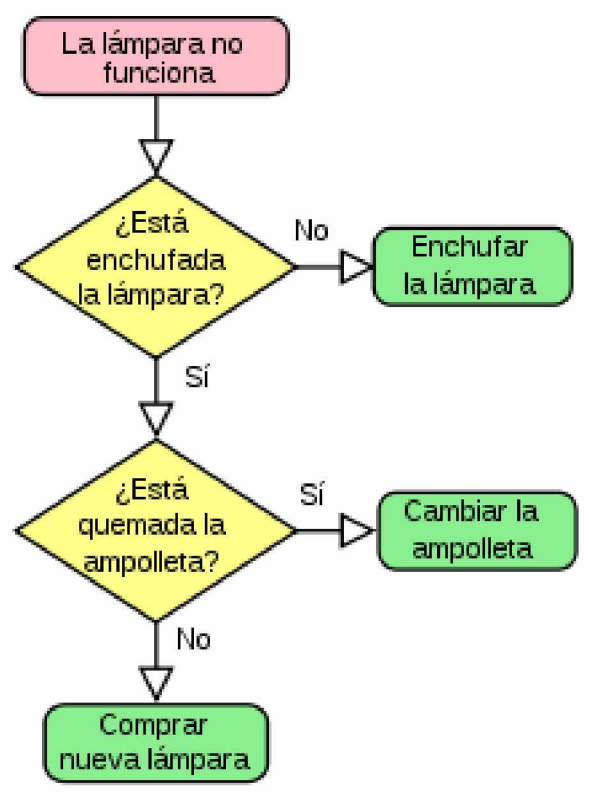
Antes de generar nuestro propio lenguaje se realizó una breve investigación para definir más claramente a que se refiere el concepto de lenguaje de programación.

Inicialmente, se entiende que un "programa" es un algoritmo; "un conjunto prescrito de instrucciones o reglas bien definidas, ordenadas y finitas que permite realizar una actividad mediante pasos sucesivos que no generen dudas a quien deba realizar dicha actividad. Dados un estado inicial y una entrada, siguiendo los pasos sucesivos se llega a un estado final y se obtiene una solución."[5]

Los algoritmos no son desarrollados necesariamente para ser ejecutados por máquinas, las instrucciones dadas por un manual de cocina dirigidas a un ser humano también se consideran un algoritmo. Y de la misma manera que para poder transmitirle a alguien las instrucciones para alguna tarea es necesario hablar el mismo idioma, para poder dar instrucciones a una máquina u ordenador es necesario un lenguaje común.

Figura 2.

Diagrama de flujo del algoritmo para hacer funcionar una lámpara.



Normalmente, cuando se habla de lenguajes de programación informático suele imaginarse una lista de palabras clave, símbolos y números escritos como texto sin formato, sin embargo no todos los lenguajes de programación se ven de esa manera. Las siguientes imágenes son diferentes programas "escritos" en los lenguajes Processing[6], PureData[7] y Piet[8], respéctivamente. Vale aclarar que la tercera imagen no es el resultado del programa, sino que es el programa en sí mismo.

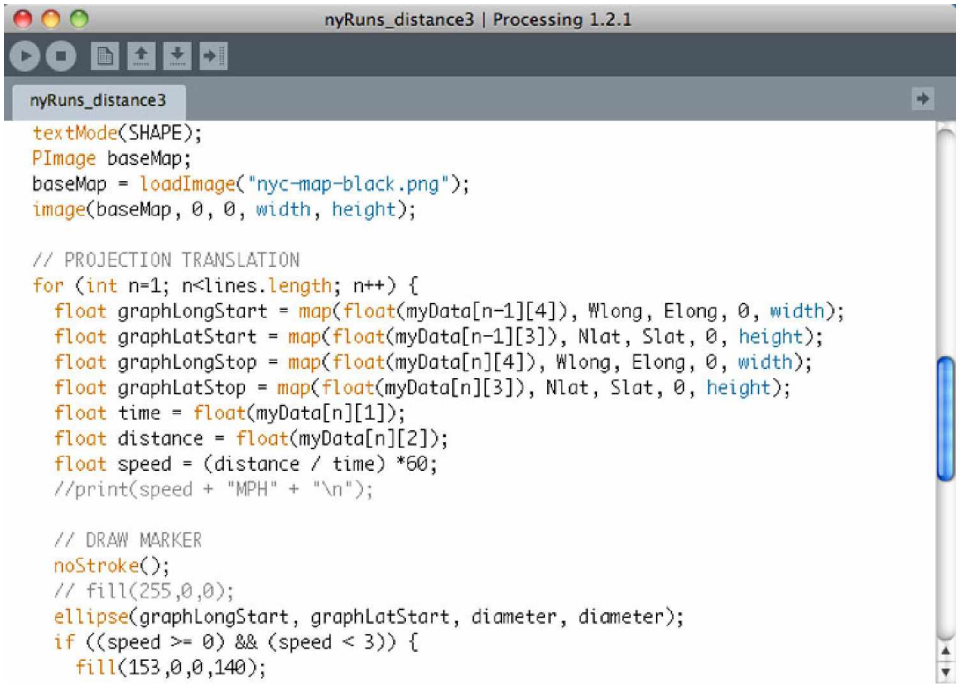


Figura 3a.

Capturas de pantalla de tres lenguajes de programación.

Figura 3b.

Capturas de pantalla de tres lenguajes de programación.

Figura 3c.

Capturas de pantalla de tres lenguajes de programación.

En todos los lenguajes de programación se encuentran dos operaciones indispensables para generar algoritmos, **estructurar y parametrizar**. Estructurar el algoritmo es definir las instrucciones que ejecutará el programa; el orden y la relación que hay entre ellas. Parametrizar es definir las variables que estas instrucciones requieren para tener un significado concreto. Para poder crear cualquier programa informático, necesitamos tener una herramienta (normalmente otro programa) que nos permita ejecutar estas operaciones en el lenguaje que estemos utilizando. Necesitamos poder ver y editar el algoritmo de alguna manera.

Teniendo esto en mente se llevó a cabo un análisis de las diferentes herramientas que permiten crear programas en diversos lenguajes, intentando tomar paradigmas variados para poder comprender que cosas eran comunes a todos ellos. Es pertinente aclarar que el análisis desarrollado no se concentra en las especificaciones técnicas de cada lenguaje -estudiadas mejor por la Ingeniería Informática-, sino en una lectura humana o conceptual de estos lenguajes, prestando mayor atención a la interfaz y lo que la interfaz propone.

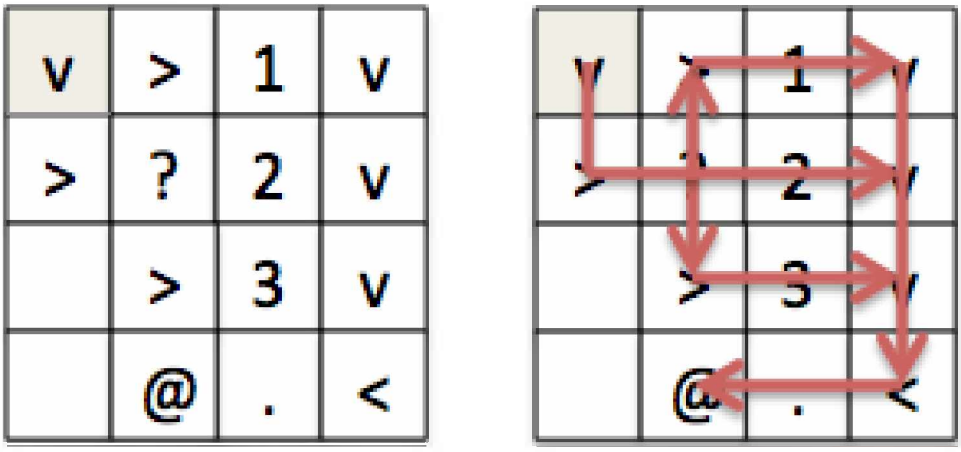
En todos los casos se pudo reconocer ciertos Elementos Generales que aparecían de manera constante, aún cuando los paradigmas de programación de cada lenguaje fueran diferentes. Estos elementos los definimos con los nombres de Materialidad, Representación, Primitivas, Parámetros, Estructuras y Espacios.

Materialidad refiere a la salida que el programa puede producir. Es común que existan herramientas que promueven ciertos tipos de salida sobre otras, pero no de manera exclusiva. Por ejemplo, tanto el lenguaje Processing como *Pure Data* pueden tener salidas en forma de video y/o audio, sin embargo el primero da mayores facilidades para ejecutar un resultado gráfico, mientras que el segundo provee facilidades para una salida sonora. Por otro lado, no dejan de existir lenguajes que están asociados a un hardware específico, y por ende, su Materialidad es limitada por el mismo, como el caso de Arduino[9], que solo puede materializar sus programa a través de sus pines electrónicos o su canal serie USB.

En contraste con la materialidad existe la Representación, refiriéndose a la representación que se le da al algoritmo que estamos trabajando. Podría decirse inicialmente que todos los lenguajes de programación tienen una representación visual, pero esto no impide que haya una gran varie-

Figura 4.

Codigo de un programa en lenguaje Befunge.



dad de representaciones, como se muestra en las imágenes anteriores. En algunos casos la representación puede ser una lista de instrucciones escritas linealmente, mientras que en otros casos puede tratarse de un mapa de flujo que describe la estructura del algoritmo. Incluso pueden encontrarse casos como el de Befunge[10] donde la representación del algoritmo consiste de un laberinto formado por caracteres.

Las Primitivas son las instrucciones y operaciones elementales que ofrece el lenguaje, podría decirse que son los ladrillos que se tienen a disposición para construir un algoritmo con él. Normalmente cuanto más sofisticadas son las primitivas de un lenguaje, más sencillo se vuelve generar programas complejos con éste. Al mismo tiempo, un lenguaje con primitivas básicas suele ser menos especializado.

Los Parámetros son la información que se provee a cada primitiva, en el caso que ésta lo permita, para que tenga una aplicacion concreta. Normalmente las primitivas que existen en un lenguaje ofrecen cierta flexibilidad durante su ejecución, dando la posibilidad de ejecutarlos dentro de cierto rango de posibilidades. En cada ejecución los parámetros se combinan con las primitivas para definir concretamente una de esas posibilidades. Por ejemplo, en Processing: consideremos la instrucción *ellipse* (indica dibujar una elipse), sus parámetros son la posicion y el tamaño que tendrá la elipse dibujada.

Las Estructuras son las diferentes formas en que podemos relacionar las primitivas del programa y la propagación (lineal, reticular, recursiva, etc.) que tiene la ejecución de un algoritmo. Cada lenguaje de programación ofrece diversas formas de estructurar primitivas y cada programa construido con ese lenguaje es una estructuración específica. Las diferentes estructuras que ofrece un lenguaje limitan y condicionan la manera en que el flujo del algoritmo se desarrolla.

Los Espacios son la distribución que presentan los elementos del programa, los "espacios" en los que pueden estar. En un lenguaje de programación es normal que la representación y la materialidad ocupen es-

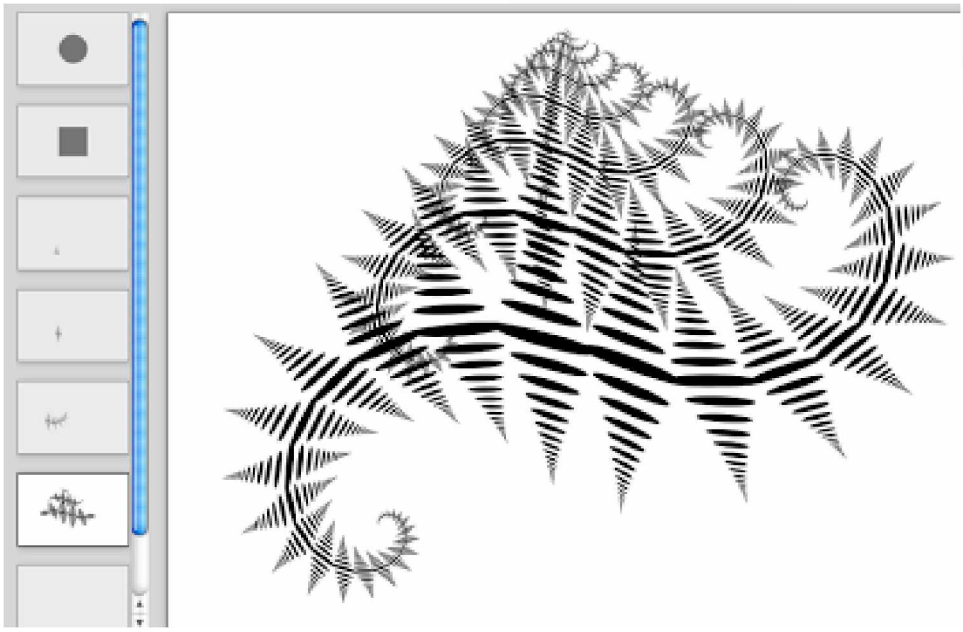
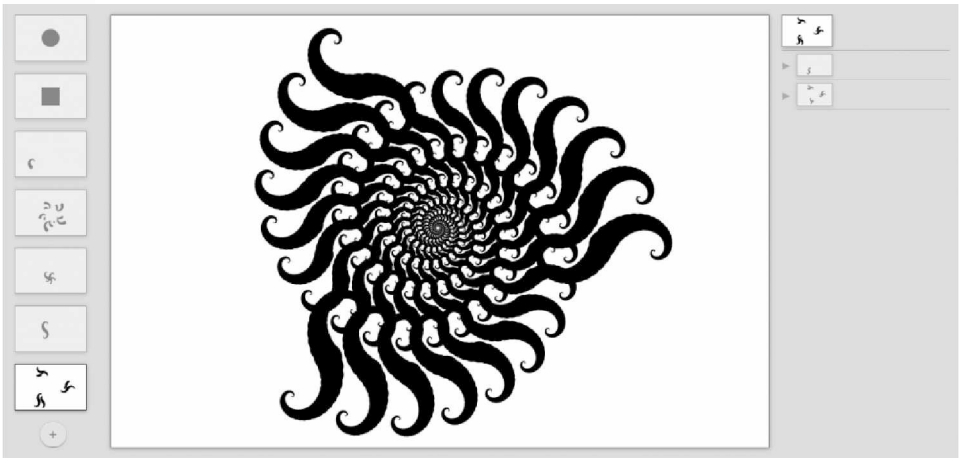


Figura 5a.

Captura de pantalla del lenguaje Recursive Drawing.

Figura 5b.

Captura de pantalla del lenguaje Recursive Drawing.



pacios diferentes, pero existen casos donde se superponen, mostrando tanto la estructura del algoritmo y la materialidad de su resultado en una misma pantalla. Incluso, en algunos casos particulares como Recursive Drawing[11] la construcción del algoritmo y su mismo resultado son difícilmente diferenciables. Sin embargo con los espacios no se hace referencia exclusivamente a la posicion espacial. Por ejemplo, existen lenguajes de programación que tienen un "momento" de ejecución y otro de edición, mientras que otros lenguajes superponen el tiempo de edición y de ejecución. De la misma forma, en el segundo caso es muy común que la posibilidad de interactuar con el algoritmo en tiempo real sea a través del mismo espacio de edición, pero en el primer caso esta interacción debe definirse previamente en el mismo algoritmo.

REACTABLE

Un caso interesante para tener en cuenta es el de Reactable[12]. Es un lenguaje de programación que posee una forma híbrida entre virtual y física. Está diseñado para generar una materialidad musical o sonora, pero la representación del algoritmo es igual de interesante. Podríamos argumentar que en este caso la representación y la materialidad no son necesariamente aspectos separados.

Otro factor interesante es que las primitivas son objetos físicos, lo que implica una serie de efectos que no suelen existir en los casos puramente virtuales. Por ejemplo, el algoritmo tiene un límite muy concreto respecto a la cantidad de primitivas que puede utilizar, y más allá de eso, las primitivas ocupan un espacio físico, lo que significa que a medida que el algoritmo crece, se obstruye a si mismo.

Esta cualidad física también es aprovechada por Reactable para trabajar otros elementos del lenguaje. Por ejemplo, los parámetros acaban siendo la posicion y rotación de cada primitiva. Otro ejemplo es que acompañando el espacio para el algoritmo activo, existe un espacio para lo inactivo. De esta manera el lenguaje utiliza los espacios como forma de indicar las primitivas que participan del algoritmo y las que no, literalmente, estando o no estando en él.

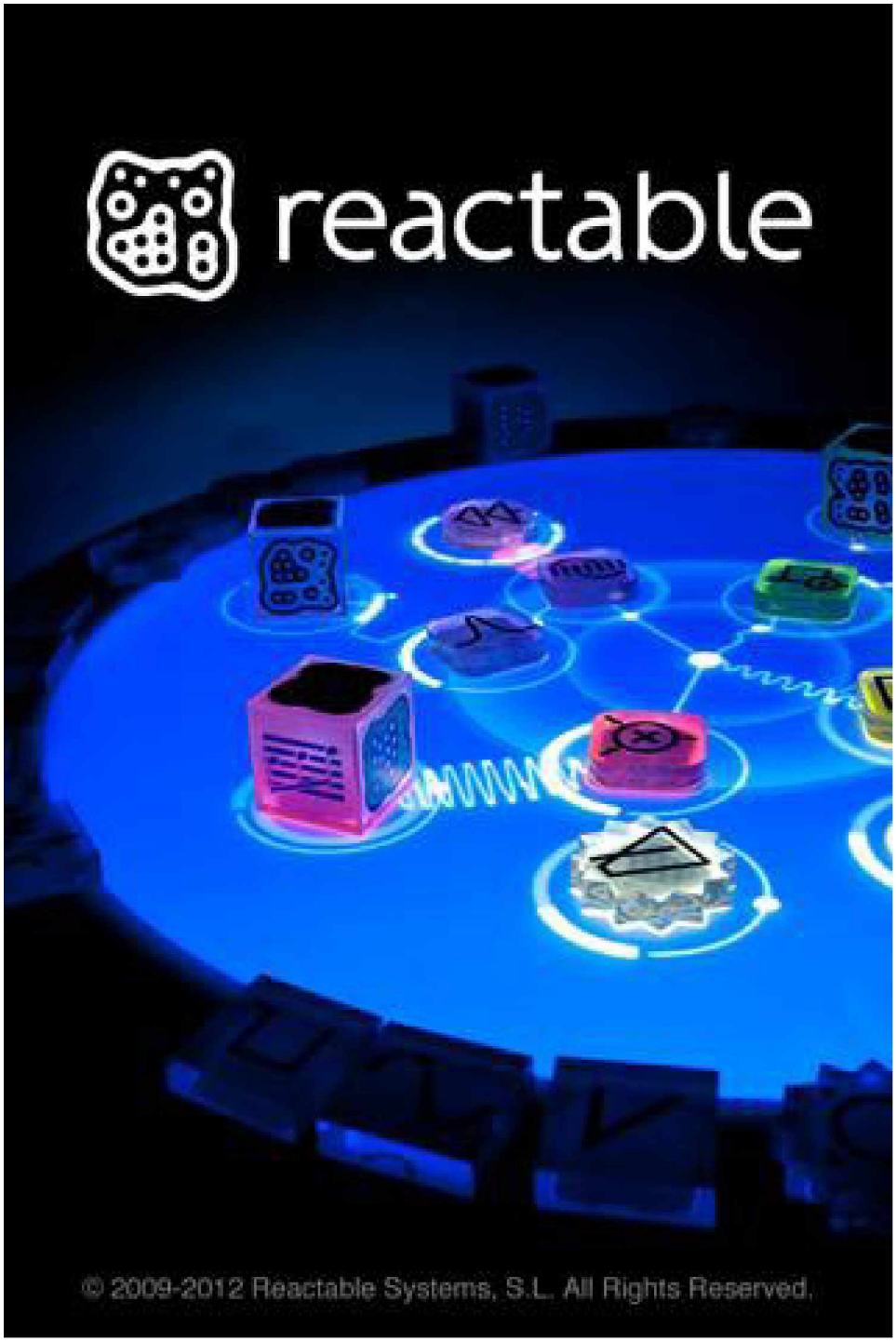


Figura 6.

Fotografia del lenguaje de programación Reactable.

APROXIMACIÓN A CREAR UN LENGUAJE

Lo más difícil a la hora de empezar a definir un paradigma de lenguaje fue evitar caer en los lugares comunes. En los análisis llevados a cabo se advirtió que cuando un lenguaje estaba diseñado con una materialidad especializada se percibía el efecto en casi todos sus elementos. Principalmente en las estructuras.También era importante tener en cuenta que la intención no era generar un lenguaje de programación general, sino llevar a cabo un experimento que permita explorar un término intermedio entre Live-Coding y Danza. Por eso se limitó fuertemente el alcance del

lenguaje, aunque sea en esta primera instancia.

Se decidió crear un lenguaje orientado a la generación de sistemas de partículas que produzcan una representación visual estética. Limitándolo a una materialidad visual dado que el EmmeLab tiene más experiencia en ese área, lo cual permitió focalizarse en el objetivo del proyecto.

Otro principio defindo para realizar este lenguaje fue que exista la posibilidad de trabajar los algoritmos con diferentes niveles de detalle, pudiendo generar diferentes sistemas de partículas de una manera más clásica (estableciendo cada elemento básico manualmente) o de una forma más general (combinando algoritmos ya creados). En este segundo caso, no sería lo usual en los lenguajes de programación (que generan estructuras cada vez más grandes), sino fusionando dos sistemas de partículas con la lógica de los algoritmos genéticos produciendo un resultado no más grande, pero sí nuevo, evitando de esta manera la complejización del algoritmo. Esto da al usuario/performer la posibilidad de elegir entre concentrarse más en el acto de generar un algoritmo, o de realizar una performance donde la realización del algoritmo sea algo secundario.

Un último factor considerado de importancia fue diseñar el lenguaje de manera que un algoritmo nunca produzca errores, y cuando el programador/performer no especifique algún parámetro necesario debe existir una solución predefinida. Es decir, el sistema debe funcionar siempre, sin importar el cuidado con que se construya el algoritmo. Esto restringió el lenguaje aún más, pero no significó problema, pues la intención no era generar una herramienta de uso general.

DISEÑO DEL SISTEMA

En una primera instancia se indagó sobre los diferentes tipos de primitivas y estructuras que deberían ser necesarias para poder generar cualquier sistema de partículas convencional. Pero rápidamente se encontraron dificultades a la hora de definir concretamente, o de cumplir con los principios inicialmente planteados.

Finalmente se tomó el camino inverso, idear cómo deberían ser los resultados del lenguaje para poder llegar a una herramienta que permita generar ese tipo de resultados.

Una de las propuestas mas considerada era que los diferentes algoritmos pudiesen fusionarse con la lógica de la combinación genética[13]. Esto quiere decir que el resultado de unir dos algoritmos no sea un algoritmo mayor que contenga a los originales, sino un algoritmo de complejidad similar, originado como resultado de la conservación de algunos de los elementos presentes en cada uno de los algoritmos originales.

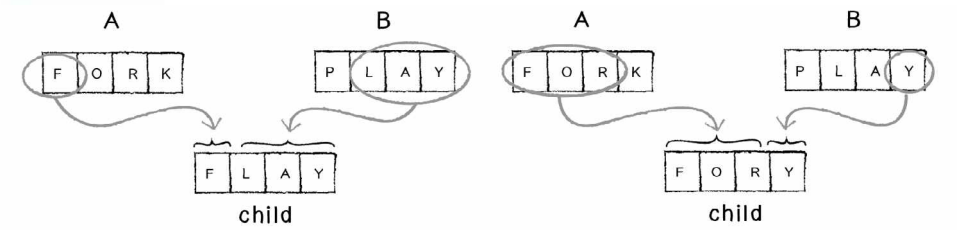


Figura 7.

Lógica de la combinación genética.

A partir de esto se propusieron cuatro sistemas de partículas relativamente variados, con sus comportamientos y representaciones. Luego, se analizó que resultados deberían producirse, o ser posible que se produzcan, al fusionar dos de ellos de la manera que planteamos previamente.

De esta manera se facilitó determinar cuáles serían las primitivas de cada algoritmo, cuáles elementos serían divisibles y cuáles no. También permitió establecer cómo se podría categorizar estos elementos.

EL SISTEMA DE PARTÍCULAS

Como la mayoría de los sistemas de partículas existentes, el sistema de partículas generado por el proyecto tiene una cantidad constante de partículas las cuales comparten un mismo comportamiento. En cambio, a diferencia de la mayoría, la cantidad y tipo de atributos que tienen las partículas son dinámicos, adaptándose a medida que generamos su programa.

Una cantidad constante de partículas significa que no se crearán o destruirán partículas durante la simulación, simplemente se define la cantidad inicial y esa cantidad se mantiene constante. Hay que tener en cuenta que a las partículas puede aplicárseles un comportamiento que *active y desactive* algunas de ellas, generando la ilusión de una cantidad dinámica de partículas.

Las partículas de un sistema son procesadas todas juntas por un solo programa, es por esto que tienen todas un mismo comportamiento. Sin embargo varios sistemas de partículas pueden convivir, generando variedad de comportamientos. Además, el sistema de partículas creado durante este proyecto fue diseñado para ofrecer la posibilidad de interacción entre varios de ellos, y para soportar condicionales. De esta manera el programa puede estar bifurcado internamente para generar comportamientos variados con un mismo sistema. Sin embargo, estas propiedades no han sido implementadas aún.

Los atributos fueron diseñados como dinámicos para crear un sistema que sea modular pero además escalable. Es decir, puede crearse un comportamiento que utilice un atributo que no exista en el sistema actual. De esta manera se abre la posibilidad a un desarrollo permanente y colaborativo de las posibilidades que ofrece el sistema. Hay que tener en cuenta que esto requeriría la creación de algún sistema que administre estas extensiones fácilmente.

GUÍA RETÓRICA

Cuando se inició con la búsqueda de la forma del lenguaje, se estableció la importancia de crear una suerte de metáfora o guía retórica. Previamente, en el momento de análisis de otros lenguajes, se advirtió que algunos de ellos parecían tener ciertas lógicas semánticas o que atravezaban todos o la mayoría de los elementos del lenguaje. Por ejemplo, en Pure Data puede verse una clara referencia a la lógica de los procesadores de audio analógicos, basados en filtros que se conectan con cables a modo de árbol, o cascada a la señal de origen. Otro caso es *Scratch*[14] que

hace referencia a los juguetes de construcción con bloques o rompecabezas, donde cada bloque tiene una suerte de bordes que por su figura sugieren donde es posible ubicarlos.

En este caso se buscó un eje retórico que posea alguna asociación con el movimiento, en un intento de generar cierta coherencia, basado en el hecho de que este lenguaje se orienta a un uso a través de la danza.

Tras el desarrollo y análisis de múltiples propuestas, se seleccionó una línea retórica inspirada en los *proyectores* fílmicos o grabadoras analógicas, donde una cinta que contiene información circula continuamente a través de mecanismos que hacen algún proceso con ella. A partir de esto se dió comienzo al diseño de una forma concreta para las primitivas y lógicas propias del sistema.

PARADIGMA GENERAL

El lenguaje se basa en dos elementos claves: una cinta o canal de información, y un sistema de mecanismos modificadores. Estos mecanismos tienen efectos relativamente concretos, relevantes, pero pueden abrirse para revelar poleas, componentes más elementales del sistema que por sí solos no generan un efecto importante. En otras palabras, las poleas se combinan para formar mecanismos. Algunos de estos mecanismos tienen el efecto secundario de producir una imagen, en cuyo caso los denominamos proyectores. Otros mecanismos funcionan enmascarando parcialmente la cinta dando la posibilidad de afectar grupos concretos de partículas.

IMPLEMENTACIÓN

Como lenguaje de base se utilizó Processing y la primera instancia a la que se abocó fue a tener un sistema de partículas que cumpla con los requerimientos definidos inicialmente aunque sea mínimamente. Luego se desarrolló una sencilla API, es decir, una forma abierta del sistema para intercambiar mensajes con otros programas. Y finalmente se escribió un programa que utiliza al dispositivo Kinect[15] e interpreta parámetros en la pose del usuario para usar como elementos de entrada al lenguaje. La ultima versión del codigo se encuentra alojada actualmente en https://github.com/emmelab2014/Nucleo.

El estado del sistema no cubre la mayoría de las características finales buscadas, como la posibilidad de fusionar dos mecanismos, la capacidad de abrir un mecanismo y modificar sus elementos básicos o la capacidad de afectar grupos específicos de partículas, pero sí cumple con los fundamentos: El sistema puede fácilmente agregar, quitar o deshabilitar mecanismos, y los atributos de las partículas son generados dinámicamente en base a las necesidades de éstos mecanismos.

Sin embargo esta prueba inicial aportó una gran cantidad de información; por un lado demostró la viabilidad del lenguaje, aparecieron los primeros problemas técnicos concretos de su realización lo cual permitió la posibilidad de reconsiderar ciertas desiciones tomadas en el primer diseño sugerido en base a datos concretos.

OBJETOS

Existen tres tipos de objetos clave en el programa actual: *Sistema, Atributo y Modificador*.

Sistema es el objeto que define un sistema de partículas específico. Contiene un ArrayList de Modificadores definido como modificadores y un HashMap de Atributos definido como atributos. Un ArrayList es simplemente una lista, los modificadores pueden repetirse en esta lista, y son ejecutados en el orden en que se encuentran. Podría decirse que es el recorrido que hace la cinta, o que es el programa del sistema de partículas. Un HashMap es una suerte de diccionario, una lista de objetos asociados a palabras claves, sin orden concreto y donde una misma palabra clave no puede aparecer dos veces. Podría decirse que este HashMap es en sí mismo la cinta.

Atributo es un objeto que sirve como esqueleto para la creación de los posibles atributos que pueden llegar a tener las partículas de un sistema, y son la información que un Modificador afecta. Cada Atributo es una extensión de este objeto padre, y todos poseen un String clave, un nombre que necesariamente debe ser único entre todos los posibles Atributos que existen en el lenguaje. Todos los Atributos deben tener un valor inicial por defecto.

Modificador es otro objeto esquemático que es extendido para generar los diferentes modificadores o procesos del programa. Actualmente el modificador está construido y tiene el comportamiento que deberían tener las poleas según el diseño inicial, pero el efecto que tienen es más semejante al que se planeó para los Mecanismos. Cada Modificador define aquellos Atributos que requiere necesariamente para funcionar, y aquellos que son opcionales.

Ejemplo General

```
void setup() {
  size(800, 600);
  sistema = new Sistema(this, 144);
  sistema.agregarModificador(mAtraccionAlCentro);
  sistema.agregarModificador(mAplicarFuerza);
  sistema.agregarModificador(mMover);
  sistema.agregarModificador(mEspacioCerrado);
  sistema.agregarModificador(mColisionParticulasSimple);
  sistema.agregarModificador(mDibujar);
}
void draw() {
  noStroke();
  fill(0,20);
  rect(0,0,width,height);
  sistema.actualizar();
}
```

En este ejemplo, se inicializa un sistema de partículas con 144 partículas, y comienzan a agregarse los modificadores que definen al programa. Luego, al ejecutar "actualizar", el sistema recorre los modificadores agregados previamente. En el primer ciclo, no posee ningún atributo, por ende, a cada paso irá creando los atributos obligatorios que requieran los modificadores. El primer modificador "mAtraccionAlCentro" requiere los atributos "Atr_Fuerza" y "Atr_Posicion", el siguiente modificador "mAplicarFuerzas" requiere solo el "Atr_Velocidad" porque el "Atr_Fuerza" ya fue insertado. Luego viene "mEspacioCerrado", que funciona con los "Atr_Posicion" y "Atr_Velocidad" pero tiene también un atributo opcional: "Atr_Tamaño". Esto significa que lo busca, pero en caso de no encontrarlo no lo agrega, y resuelve su ejecución con información fija, predeterminada por el mismo modificador. Los modificadores consecutivos son "mMover" que no necesita más atributos de los que ya están y "mColisionParticulasSimple" que necesita obligatoriamente al "Atr_Tamaño", asique finalmente es un atributo que entra al sistema. Por último se encuentra "mDibujar" que tampoco necesita nuevos atributos.

Finalmente la cinta acaba teniendo los atributos Fuerza, Posicion, Velocidad y Tamaño. Estos atributos van siendo procesados por los modificadores en el orden que fueron asociados.

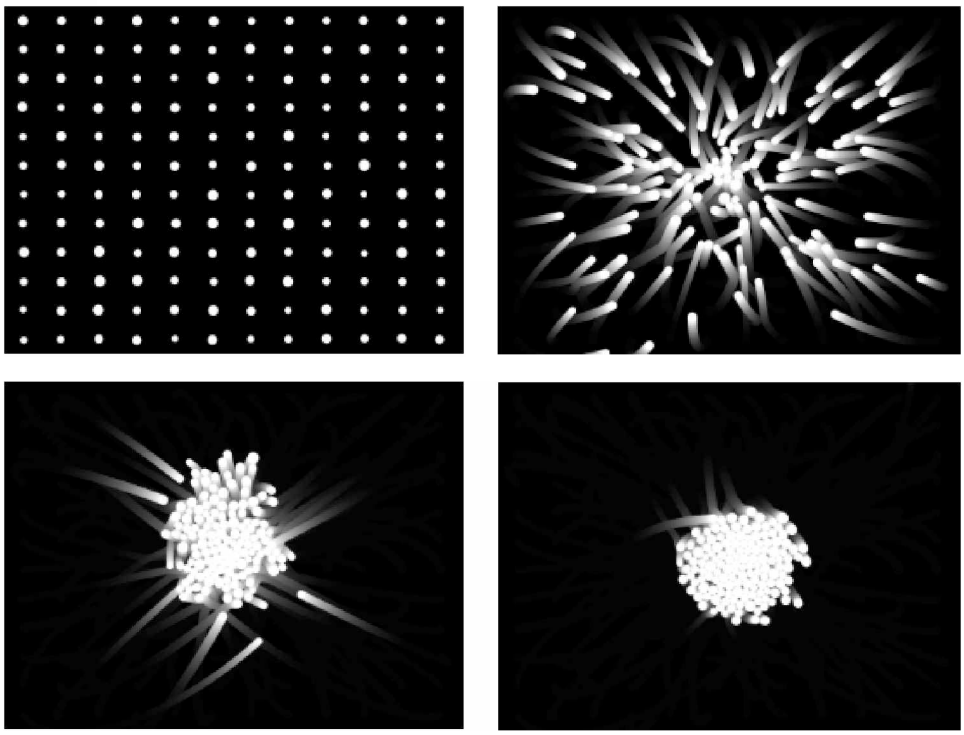
Primero, "mAtraccionAlCentro" ejerce una fuerza sobre las partículas en dirección al centro, luego "mAplicarFuerza" transforma esta fuerza en velocidad, seguido por "mMover" que actualiza la posición en base a la velocidad.

"mEspacioCerrado" simplemente hace que las partículas reboten contra los bordes de la ventana, pero en este caso su efecto no se manifiesta.

"mColisionParticulasSimple" les da cuerpo a las partículas, evitando que se superpongan, haciendo que choquen y reboten entre sí. Si este modificador no existiera, las partículas se solaparían.

Figura 8a-8b-8c-8d.

Resultado visual del ejemplo anterior

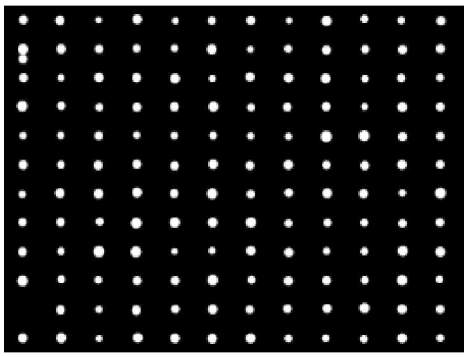


Finalmente "mDibujar" representa las partículas como un circulo tomando en cuenta la Posicion, Tamaño, y, si lo tubiera, el Color. El efecto de estela no es propio del sistema de partículas, sino que se da a causa de las primeras tres líneas de código en la función "draw".

Otros Ejemplos:

Ejemplo 1

```
void setup() {
  size(800, 600);
  sistema = new Sistema(this, 144);
  sistema.agregarModificador(mGravedad);
  sistema.agregarModificador(mResetLluvia);
  sistema.agregarModificador(mMover);
  sistema.agregarModificador(mEspacioCerrado);
  sistema.agregarModificador(mDibujar);
}
void draw() {
  noStroke();
  fill(0,20);
  rect(0,0,width,height);
  sistema.actualizar();
}
```



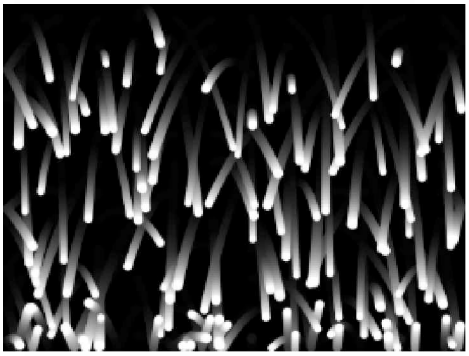
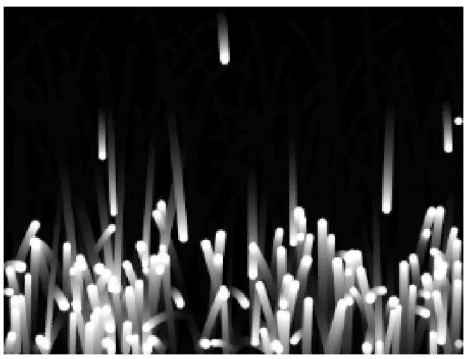


Figura 9a-9b.

Resultado visual del ejemplo anterior



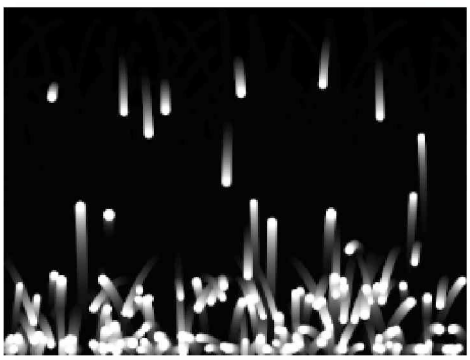


Figura 9c-9d.

Resultado visual del ejemplo anterior

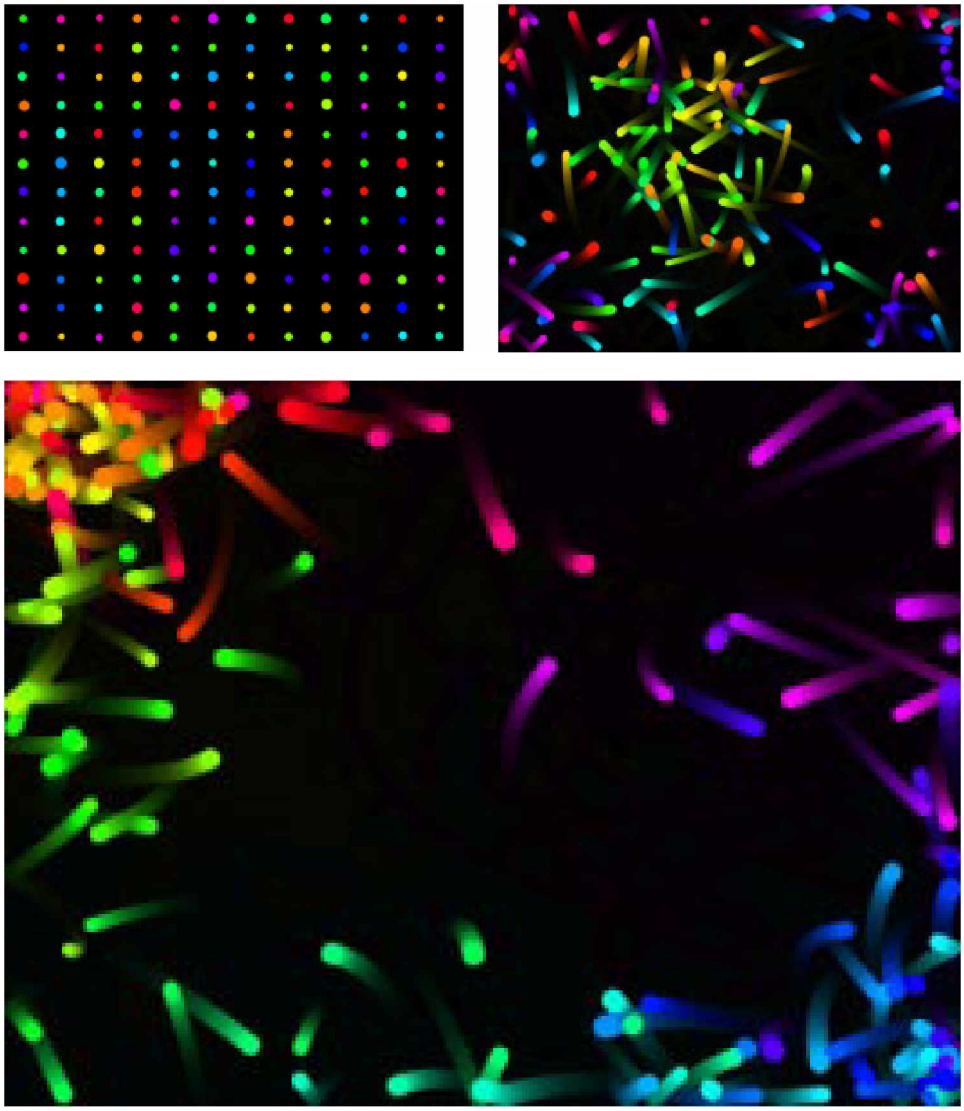
El modificador "mResetLluvia" hace que las partículas reaparezcan arriba cuando estan en el suelo sin velocidad.

Ejemplo 2

```
void setup() {
  size(800, 600);
  sistema = new Sistema(this, 144);
  sistema.agregarModificador(mFuerzasPorSemejanza);
  sistema.agregarModificador(mAplicarFuerza);
  sistema.agregarModificador(mMover);
  sistema.agregarModificador(mEspacioCerrado);
  sistema.agregarModificador(mDibujar);
}
void draw() {
  noStroke();
```

Figura 10a-10b-10c.

Resultado visual del ejemplo anterior



```
  fill(0,20);
  rect(0,0,width,height);
  sistema.actualizar();
}
```

Ejemplo 3

```
void setup() {
  size(800, 600);
  sistema = new Sistema(this, 144);
  sistema.agregarModificador(mAtraccionUniversal);
  sistema.agregarModificador(mAplicarFuerza);
  sistema.agregarModificador(mMover);
  sistema.agregarModificador(mColisionParticulasSimple);
  sistema.agregarModificador(mEspacioToroidal);
```

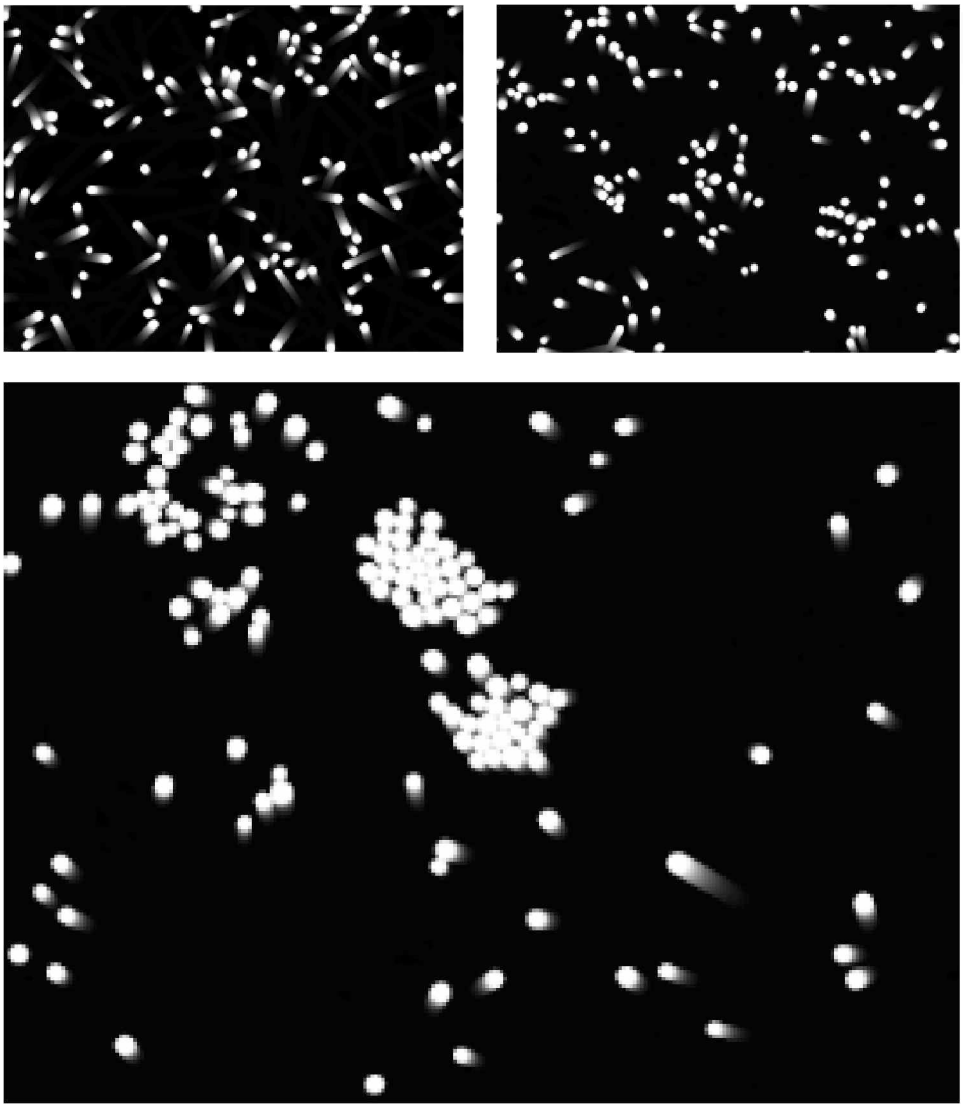


Figura 11a-11b-11c.

Resultado visual del ejemplo anterior

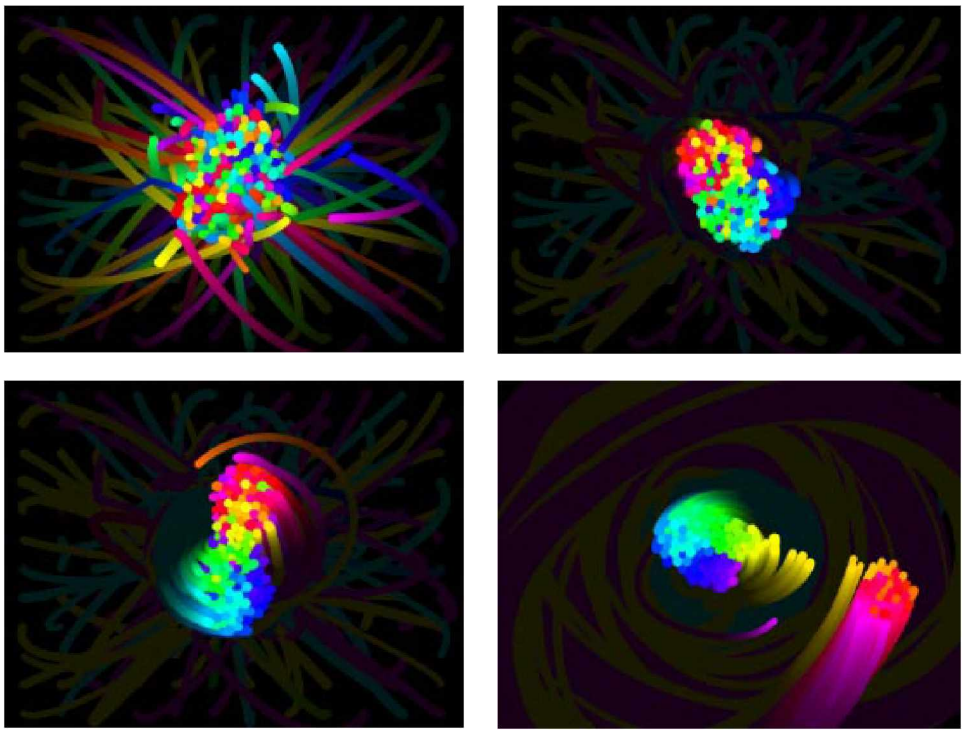
```
  sistema.agregarModificador(mDibujar);
}
void draw() {
  noStroke();
  fill(0,20);
  rect(0,0,width,height);
  sistema.actualizar();
}
```

Ejemplo 4

```
void setup() {
  size(800, 600);
  sistema = new Sistema(this, 144);
  sistema.agregarModificador(mFuerzasPorSemejanza);
```

Figura 12a-12b-12c.

Resultado visual del ejemplo anterior



```
  sistema.agregarModificador(mAtraccionAlCentro);
  sistema.agregarModificador(mAplicarFuerza);
  sistema.agregarModificador(mMover);
  sistema.agregarModificador(mColisionParticulasSimple);
  sistema.agregarModificador(mDibujar);
}
```

```
void draw() {
  noStroke();
  fill(0,5);
  rect(0,0,width,height);
  sistema.actualizar();
}
```

Notas

1. http://toplap.org/
2. http://networkmusicfestival.org/nmf2012/programme/performances/silicone-bake/
3. http://rtigger.com/blog/2013/05/15/live-coding-good-or-bad
4. http://www.lighthouse.org.uk/programme/brighton-digital-festival-closing-performances
5. http://es.wikipedia.org/wiki/Algoritmo
6. https://processing.org/
7. http://puredata.info/
8. http://www.dangermouse.net/esoteric/piet.html
9. http://www.arduino.cc/
10. http://es.wikipedia.org/wiki/Befunge
11. http://recursivedrawing.com/
12. http://www.reactable.com/
13. http://natureofcode.com/book/chapter-9-the-evolution-of-code/
14. https://scratch.mit.edu/
15. http://es.wikipedia.org/wiki/Kinect