Live Coding

El paradigma de la programación en vivo

v.2

Hernani Villaseñor Ramírez, 2013
Centro Multimedia, CENART, México D.F.

¿Qué es *Live Coding*?

Live Coding es una práctica derivada de la música electrónica por computadora y la video animación donde se programa en vivo mientras se expone el código junto con el sonido y la imagen resultante.

Esta corriente comenzó a tomar fuerza en Europa en los años 90 y actualmente es un ejercicio dentro del arte electrónico experimental. McLean (129), menciona que el término surgió alrededor del 2003, para describir una actividad con una aproximación a nuevas formas de hacer música por computadora y video animación, asimismo sugiere que el término *live coding* se usa más en el contexto de la improvisación.

Por su parte, Rohrhuber *et al* (2002), mencionan que *"Just in time programming (or: conversational programming, live coding, on-the-flying-programming, interactive programming)"* se refiere a un paradigma que incluye la actividad de programar en la funcionalidad misma del programa, lo que hace que escribir código se convierta en una actividad cercana a una práctica musical o experimental.

Antecedentes

The Hub es un ensamble de programadores/compositores pioneros en la práctica de Música por Computadora en Red[1]. Este ensamble se caracteriza por compartir datos e información a través de una red para ser modificados durante la presentación. El ensamble

1 Computer Music Network

esta formado por Tim Perkins, John Bischoff, Chris Brown, Scot Gresham-Lancaster, Mark Trayle y Phil Stone.

Slub es un ensamble originalmente formado en Londres por Alex McLean y Adrian Ward, quienes diseñan sus programas de audio en base a lenguajes como Pearl y REALBasic (Collins et al, 2003:323). Posteriormente, Dave Griffiths se unió a este ensamble de música por computadora.

PowerBooks Unplugged es un ensamble formado en el 2003 por 6 músicos que usan la computadora como un instrumento autónomo, aprovechando la capacidad de conexión inalámbirca. Formado por Alberto de Campo, Echo Ho, Hannes Hoelzl, Jan-Kees van Kampen, Julian Rohrhuber y Renate Wiser. PB_UP tocan con sus laptops sin conectarse a un sistema de sonido, intercambiando código a través de una red, el cual van generando y modificando en el momento de la presentación.

aa-cell es un duo australiano de *live coding* activo desde el 2005, conformado por Andrew R. Brown y Andrew Sorenson, quienes usan el programa Impromptu en sus presentaciones, el cual ha sido diseñado por Sorenson.

Benoît and the Mandelbrots, es un ensamble de *live coding* formado en el 2009 en Karlsruhe. Conectados a través de una red se sincronizan y envian datos. Integrado por Juan A. Romero, Patrick Borgeat, Holger Ballweg y Matthias Schneiderbanger.

También podemos mencionar los primeros festivales de *live coding* “Changing Grammars” organizado en el 2004 en Hamburgo y “LOSS Livecode” en el 2007 en Sheffield.

Y finalmente la organización TOPLAP (Temporary Organisation for The Promotion of Live Algorithm Programming) un grupo de *live coders* establecido para discutir y promover el *live coding* (McLean 2011:131).

Live Coding en México: una perspectiva desde el Centro Multimedia

La práctica de *live coding* en el Centro Multimedia tiene sus orígenes en el año 2006, con una serie de conciertos de la agrupación mU, la cual estaba formada por Eduardo Meléndez, Ezequiel Netri y Ernesto Romero quienes trabajaban en el Taller de Audio del Centro Multimedia, estos conciertos tuvieron lugar en la Universidad Iberoamericana, el Festival Plataforma de Puebla y Dorkbot México. Además de este evento, el 14 de julio 2009, se llevó acabo el primer concierto de Sinescenia[2], donde Ezequiel Netri presentó "Prácticas con código en vivo: Re-modulación en vivo de código abierto a través de Supercollider"*,* que junto a los conciertos de mU podrían considerse las primeras presentaciones de de *live coding* en el CMM[3].

En diciembre 2010, con una creciente comunidad de usuarios de SuperCollider, Fluxus y Processing cercanos al CMM, se dan las condiciones para hacer la primer sesión de *live coding* con varios participantes, la cual se llevó acabo en la Galería de arte electrónico Manuel Felguérez del CMM. Así, a partir del 2011, el Taller de Audio del CMM organiza mensualmente una sesión de *live coding*[4].

En el CMM esta práctica se ha difundido a partir de la programación de sonido e imagen. Los programas más usados son SuperCollider para sonido y Fluxus y Processing para gráficos, aunque también se programa en Max/MSP, Pure Data, OpenFrameworks, VVVV, y en ocasiones lenguaje C directamente sobre microcontroladores.

En un inicio estas sesiones contaron con mucho público, aunque poco a poco se convirtieron en meras reuniones de entusiastas de la programación. Entonces, a partir de la sesión 14 se cambió la estrategia de organización, y cada sesión comenzó a realizarse en una sede distinta al CMM, atrayendo así, nuevos públicos. Lugares como SAE Institute, Fonoteca Nacional, Ciencias UNAM, el Laboratorio de Video en la Esmeralda, Casa Vecina y

2 Sinescenia es una colaboración que se ha desarrollado por 3 años entre el Centro Multimedia del CENART y el Centro Cultural de España en México y se enmarca dentro del programa de arte y nuevos medios del CCE.

3 Centro Multimedia del Centro Nacional de las Artes. México.

4 Hasta el 20 de octubre 2012 se habían realizado 20 sesiones de Live Coding en el CMM, promovidas por el Taller de Audio.

el Museo de Ciencias de Morelos han sido nuevos escenarios para estas sesiones, que poco a poco se han establecido como una plataforma para exponer el trabajo de artistas y programadores, quienes consolidan la práctica en México.

Más allá de las sesiones de Live Coding en el CMM

La sesiones de *live coding* en el CMM, además de servir como plataforma para difundir una práctica y crear una comunidad, han sido detonador para la organización de otros eventos ligados a esta práctica, por ejemplo “Diarios Efímeros” ciclo organizado por Jaime Lobato dentro de IndexMUAC, programa de música contempránea del Museo Universitario de Arte Contemporáneo de la UNAM. Este ciclo se llevó a cabo entre el 16 de agosto y el 30 septiembre 2012, con tres actos en vivo de *live coding* sumado a otras disciplinas como pintura, danza y *circuit bending*[5].

Como figuras destacadas de la comunidad en torno a las sesiones de *Live Coding* del CMM se puede mencionar a Ernesto Romero que junto a Hernani Villaseñor han organizado y promovido esta práctica dentro de este espacio. También podemos mencionar a Luis Navarro, Eduardo H Obieta, Jorge Ramírez, Martín Zumaya, Jaime Lobato, Alejandro Franco, Mitzi Olvera, Alexandra Cárdenas, Emilio Ocelotl, Jaime Jalil Ramírez, José Carlos Hasbun, Julio Zaldívar, Arsitotles Benitez y el Colectivo Radiador.

/*vivo*/ a dos años de trabajo sobre el tema

En el 2012 se organizó el Simposio Internacional de Música y Código /*vivo*/, el cual tuvo lugar en el CMM entre el 13 y 16 de noviembre 2012 con el tema: *Live Coding,* justo dos años después de haber comenzado la sesiones mensuales. Este simposio fue el tercero realizado a nivel mundial con esta temática y contó con la participación de Julian Rohrhuber, Alex McLean, Dave Giffiths, Johannes Zmölnig y Alberto de Campo.

5 Modificación de juguetes de pila.

Introducción a SuperCollider

SuperCollider es un lenguaje de programación de sonido, en el cual se programa código de la misma forma en que escribimos en un procesador de texto. Este programa fue diseñado por James McCartney en 1998, y desde entonces ha sido adoptado por una comunidad que trabaja en torno a la música algorítmica y el sonido creado por computadora.

Conoce tu teclado

Encuentra en tu teclado los siguientes caracteres: ~, [ ], { }, | |. Tilde, corchete, llave y *pipe*. Encuentra las siguientes funciones: enter, cmd, alt, ctrl, esc.

Hola Mundo !

Como en cualquier aproximación al estudio de una herramienta de programación, el primer paso es asegurarnos que está instalada correctamente y que funciona, esta primer programación se llama "*Hello World !"*

Para realizar nuestro "Hola mundo !" en SuperCollider necesitamos saber como activar una línea de código y como frenar el proceso. Estas acciones están determinadas por el sistema operativo que usamos y por la forma en la que corremos SuperCollider[1]. Entonces, para hacer funcionar un código ponemos el cursor sobre la línea que queremos activar o seleccionamos varias líneas, por último presionamos la tecla que nuestra versión de SuperCollider requiere. Para frenar el proceso basta presionar las teclas que accionan el comando *stop.* Nuevamente, este *shortcut* es distinto en cada sistema.

A continuación los *shortcuts* en los sistemas más comunes:

1 Por ejemplo SuperCollider puede funcionar dentro del editor de textos Gedit en Linux.

Sistema	Encender	Parar
Mac	enter	cmd + .
Windows	ctrl + intro	ctrl + .
Linux - Gedit	ctrl + e	esc
Linux – Emacs	C-c C-c	C-c C-s

Ahora que sabemos cómo declarar código y como frenar procesos, intentemos con el siguiente ejemplo, el cual debemos declarar línea por línea, el resultado será un tono sinoidal de 440 Hz en la bocina izquierda:

```
s.boot;
{SinOsc.ar}.play
```

Una vez que hemos probado que la versión de SuperCollider funciona en nuestra computadora procederemos a revisar la sintaxis básica del programa.

Ver ejemplo 01 en la carpeta de ejemplos.

Sintaxis básica

Para escribir código en SuperCollider es necesario saber que significan los siguientes caracteres en el contexto de la programación, a continuación definimos aquellos con los que iremos construyendo un código funcional.

//	comentario de una línea
/* */	comentario de un párrafo extenso
{ }	engloba una función
[ ]	engloba un arreglo
( )	engloba argumentos de un objeto
| |	engloba argumentos declarados

SinOsc	objeto, siempre empieza con mayúscula
.play	método, lleva un punto antes del mensaje
;	ruptura de código
,	una coma separa los argumentos
“una palabra”	comillas, engloban texto dentro de un código en función
\nombre	diagonal, define el nombre de una rutina, un sinte o un mensaje
~	tilde, define una variable global

Todos estos caracteres no tiene sentido por si solos, pero es necesario identificar que función tienen dentro del código, esto en un principio es importante ya que el código podría parecernos una secuencia de palabras y números incomprensibles. Al definir estos caracteres podemos leer código sin perdernos y detectar los errores más rápido.

Ver ejemplo 02 en la carpeta de ejemplos.

La ventana del servidor local

SuperCollider tiene dos servidores: local e interno. En nuestro caso trabajaremos con el servido interno. La ventana del servidor interno nos muestra el comportamiento de SuperCollider, de interés son: la cantidad de procesamiento empleado y el número de sintetizadores activos. Desde esta ventana también podemos prender nuestro servidor y activarlo como *default*.

La ventana *post*

Esta ventana nos indica con mayor precisión lo que sucede con SuperCollider, nos informa que el programa se ha encendido correctamente, la característica de cada función que declaramos y nos alerta cuando hay un error en la programación. También imprime la información que le indiquemos o los valores arrojados por una acción.

```
// declara este código en supercollider
```

```
12.postln
"imprime esto".postln
```

Lectura de errores

Como mencioné antes, la ventana *post* nos alerta de errores en la programación, esto es de suma importancia cuando practicamos *live coding*, ya que cuando algo no suena es buena idea revisar si hay un error en la *post*, de esta manera tendremos elementos para resolverlo.

Estas alertas nos indican en que línea esta el error y de que tipo es, a veces nos da la suficiente información para resolverlo, otras tenemos que revisar directamente el código que acabamos de declarar y que arrojó un error.

UGens

La unidades generadoras -*Unit Generators*-, son pequeños módulos de programación que aceptan parámetros de entrada con al menos una salida. Estas unidades generadoras pueden conectarse a otras para diseñar, lo que en música electrónica, se considera según Dodge y Jerse(1997:72), un instrumento o algoritmo el cual realiza un evento musical.

En SuperCollider estas unidades se llaman UGens y son las encargadas de generar y controlar sonido. Los UGens aceptan dos tipos de mensajes uno de audio .ar y uno de control .kr y estan compuestos por argumentos, que son parámetros susceptibles de ser modificados. Así, el UGen que es capaz de generar un tono puro se llama SinOsc, el cual contiene los argumentos de frecuencia, fase, multiplicación (amplitud) y adición, a su vez, este UGen acepta los mensajes de audio y de control.

```
//un tono puro de 440, con fase 0, la amplitud máxima,y la adición en 0
SinOsc.ar(440,0,1,0)
```

Entonces en el ejemplo anterior observamos que un UGen u objeto (SinOsc) en SuperCollider empieza con mayúscula, después se coloca el mensaje el cual comienza con minúscula (.ar), luego se abren unos paréntesis y dentro están los argumentos. Cada UGen tiene argumentos distintos los cuales se especifican en un documento de ayuda.

Cabe mencionar que en SuperCollider la amplitud se expresa mediante una multiplicación con un valor mínimo de 0 y un valor máximo de 1, esto es lo que se conoce como amplitud normalizada.
UGens generadores de sonido:

SinOsc, Pulse, Saw

UGens de baja frecuencia diseñados para modular (LFO):

LFPar, LFTri, LFPulse, LFCub, Impulse, LFSaw, VarSaw, SyncSaw

Ruidos:

WhiteNoise, PinkNoise, Dust, LFNoise0, LFNoise1, LFNoise2

Variables

Son una especie de contenedor reservado en la memoria de la computadora que guarda un valor. Las variables representan algo y son definidas por el usuario.

En SuperCollider existen las variables y las variable globales. Las variables tienen que ser declaradas dentro de un fragmento de código antecedidas por la abreviatura var. Las variables declaradas de esta manera deben empezar con minúscula y solo afectan un fragmento de código.

```
{ var sin, pulso;
```

```
sin = SinOsc.ar(200,0,0.1);
pulso = Pulse.ar (0.5);
sin * pulso}.play
```

Las variables globales son letras minúsculas o palabras que empiezan con una tilde como ~variable. Las variables globales pueden ser declaradas en cualquier momento para ser usadas cuando queramos. Es importante mencionar que hay letras que son reservadas como la 's' que esta asignada para *Server*.

```
a = 5
b = 7
a + b

~frecuencia = 500
~amplitud = 1
```

Ver ejemplo 03 en la carpeta de ejemplos.

Argumentos

Los argumentos en SuperCollider son valores asignados a cada objeto, los cuales tienen un rango y están definidos dentro de la programación de cada objeto. Para saber cuantos argumentos tiene un objeto, qué define cada uno y cuál es su rango de operación, basta ver la ayuda del objeto. En Mac seleccionamos el texto del objeto y presionamos cmd + D.

Analicemos el objeto que genera tonos sinoidales: SinOSc.ar, el cual tiene 4 argumentos que son: frecuencia, fase, multiplicación y adición.

SinOsc.ar (freq, phase, mul, add)

En este caso *freq* dice que frecuencia genera el oscilador, *phase* en que momento

comienza la onda, este argumento se mide en radianes; luego *mul,* es la abreviatura de multiplicación lo cual define la amplitud normalizada en un rango de 0 – 1, y por último *add* es una suma la cual se puede usar para controlar el comprtamiento del oscilador.

```
// así viene especificada la ayuda de SinOsc
SinOsc.ar(freq, phase, mul, add)
```

Arrays

Los arreglos o *arrays* son contenedores de información, son elementos organizados en una lista, generalmente números que pueden usarse de maneras diversas. En SuperCollider un arreglo se encasilla entre corchetes y sus elementos se separan por comas, cada elemento es un índice y tiene una posición dentro del *array* con la cual puede ser accesado. Así, el primer elemento es el índice 0, el segundo índice 1, etc.

```
// array de 9 valores
[1,2,3,4,5,6,7,8,9]
```

Los valores internos de los *arrays* se identifican mediante índices, comenzando por el índice 0.

```
// array de 9 valores, 9 es el índice 0, 3 el índice 1, 4 el índice 2, etc.
[9,3,4,7,8,2,3,5,6]
```

Para acceder a un índice del *array* utilizamos un operador de acceso que en SuperCollider funciona indicando entre corchetes el número de índice que requerimos, antes debemos igualar el *array* a una variable.

```
// igualamos una array de 9 elementos a la variable 'a'
a=[3,4,7,9,3,2,4,1,0]
```

```
// el operador de acceso llama el índice que le indicamos
a[0]

// la post nos arroja el número 3, que es el valor que ocupa la primera
posición del array
```

Los arrays también aceptan otro tipo de valores como puede ser una lista de palabras. Además son flexibles ya que en realidad cada elemento del array es una variable, es decir es susceptible de ser sustituida por otro valor.

Ver ejemplo 04 en la carpeta de ejemplos.

Live Coding con SuperCollider: La librería JITLib

JITLib es una librería de SuperCollider desarrollada por Julian Rohrhuber. JITLib significa "*Just in time programming"* otra forma de referise al *live coding*.

ProxySpace

ProxySpace pertenece a la librería JITLib y es uno de los cuatro entornos de esta librería con el cual es posible modificar el código de una programación mientras corre.

Primeros pasos

Para comenzar iniciamos el ambiente Proxy declarando la siguiente línea de código:

```
p = ProxySpace.push(s.boot)
```

Luego declaramos una variable global, para lo cual anteponemos una tilde a la palabra que formará nuestra variable y usamos el mensaje .play, de esta manera hemos creado algo

que suena, aunque todavía no existe, este es el concepto de *Proxy*, un contenedor abstracto de datos.

```
~salida.play
```

A continuación igualamos nuestra variable a una función que contenga un UGen o elemento sonoro, por ejemplo un oscilador de onda triangular:

```
~salida={LFTri.ar(400, 0, 0.5)}
```

De esta manera suena la frecuencia de 400 Hz de una onda triangular y estamos listos para modificar nuestro primer código dentro de *ProxySpace*.

Ver ejemplo 05 en la carpeta de ejemplos.

Play/Stop

Los mensajes .play y .stop sirven para iniciar un proceso y para detenerlo, de esta forma al mandar el mensaje .play el proceso inicia de golpe. Algo similar sucede al frenar el proceso mediante el mensaje .stop, ya que este detiene inmediatamente su acción.

```
// prendo y apago los procesos de un sinte, este se libera del server
~a.play
~a={SinOsc.ar(400,0,0.5)}
~a.stop
```

Pause/Resume

Pone en pausa un sinte recién declarado, para continuar escuchandolo usamos .resume, de esta manera el sonido se silencia de manera inmediata, con .ar puede producir *clicks.* Cuando usamos .pause el sinte se mantiene activo dentro del servidor, podemos modificarlo mientras y sus cambios tomaran efecto cuando este sea reanudado.

```
// pongo en pausa y continuo los procesos, el sinte no se librea del server
~a.play
~a={SinOsc.ar(400,0,0.5)}
~a.pause
~a.resume
```

Release/Send

Estos mensajes no afectan el monitoreo del sinte, .send lo que hace es encender un nuevo sinte, .release lo libera. Observemos como el sinte sale del Server. En este caso fadeTime afecta el tiempo de encendido y de liberación.

```
// libero y pongo en marcha un sinte, el sinte se librea del server
~a.play
~a={SinOsc.ar(400,0,0.5)}
~a.release
~a.send
```

FadeTime

fadeTime determina el tiempo que utilizará cada nuevo cambio, podemos declararlo de manera general o asignarlo a cada variable.

Hemos igualado nuestro ambiente Proxy a 'p', así que podemos crear un fadeTime global de 10 segundos de la siguiente manera:

```
p.fadeTime(10)

// cada cambio responderá al tiempo asignado a esta variable
```

```
~salida.fadeTime=10
```

Ver ejemplo 06 en la carpeta de ejemplos.

Cambio de valores con el método set y xset

El método .set se usa para cambiar el valor asignado a un argumento, por lo que es necesario llamarlo a través de la variable que representa un sinte. Este método se coloca después de la variable, y entre parentesis se pone el nombre del argumento precedido por una diagonal, después de una coma el nuevo valor. Una vez hecho el cambio se declara y el resultado tiene efecto.

```
// freq es nuestro argumento el cual lo igualamos a 400
~y={|freq = 400| SinOsc.ar(freq, 0, 0.5)}.play

// posteriormente con el método.set sustituimos el valor 400, por 800
~y.set(\freq, 800)
```

El método .xset, genera un *crossfade*[6] que depende del valor asignado en fadeTime, es decir si este es de 5 segundos la transición que se logra entre el valor anterior y el nuevo es de 5 segundos, esto crea transiciones que evitan los cambios abruptos al sustituir los valores de un sonido con el mensaje .set.

```
// este método es similar al anterior, la diferencia es que aplica un
crossfade que depende de fadeTime
~y.xset(\freq, 900)
```

Ver ejemplo 07 en la carpeta de ejemplos.

Control y mapeo

6 Cruce de señales mediante una transición de tiempo.

Es posible declarar las funciones de manera explícita, es decir, indicar si serán de control o de audio, para posteriormente utilizarlas de manera anidada dentro de otra función.

```
// especifíco que la variable será de control y asigno el control mediante el ratón
~algo.kr
~algo={MouseX.kr(200,2000)}

// a un SinOsc le añado como argumento la variable ~algo.kr que controlará su frecuencia
~objeto.play
~objeto={SinOsc.ar(~algo.kr,0,0.5)}
```

Es posible mapear el control mediante el método .map utilizando argumentos dentro de la construcción del sinte.

```
~a.play
// creamos el argumento frec
~a={|frec=400| SinOsc.ar(frec, 0, 0.5)}

// creamos la variable ~c y le asignamos un control con el ratón
~c={MouseX.kr(300, 3000)}

// con el método .map indicamos que el contenido de ~c modifique el argumento \frec
~a.map(\frec, ~c)
```

Para quitar el mapa usamos el método .unmap sobre la variable que fue mapeada anteriormente.

```
// ambos métodos remueven el mapeo
~a.unmap
~a.uset
```

Una variante de .map con la que podemos lograr transiciones suaves es .xmap, este método utiliza el .fadeTime que es aplicado sobre la variable.

Ver ejemplo 08 en la carpeta de ejemplos.

Aleatoridad y reconstrucción de valores

Hay varios métodos para crear valores aleatorios en SuperCollider, por ejemplo rrand o exprand, que indica que escogerán un número de manera aleatoria dentro de un rango que nosotros indiquemos.

```
// cada vez que declaremos el sinte se producirá un nuevo valor entre 100 y 200
~a.play
~a={SinOsc.ar(rrand(100,200),0,1)}
```

Para reconstruir estos valores podemos mandar el mensaje .build, el cual sirve para reconstruir un sinte dentro del servidor, es bueno usarlo si el sinte fue programado con valores aleatorios.

```
// le dice al sinte que reconstruya los valores aleatorios
~a.rebuild
```

Ver ejemplo 09 en la carpeta de ejemplos.

Generación de ritmo mediante Pulsos

Hay varios métodos en SuperCollider que generan pulsos, este recurso es socorrido para generar rítmicas dentro de una programación, y puede utilizarse como una estrategia para realizar programación en vivo. Esta técnica nos permite tener resultados mediante un pulso que sirve como detonador de ritmos, aunque no es tan flexible en la programación de rítmicas como podría ser el uso de patrones. Lo que hacemos es multiplicar un UGen por un pulso.

Veamos dos casos de UGens que producen pulsos: LFPulse y LFNoise0.

```
~a= {SinOsc.ar(200,0,0.5) * LFNoise0.ar(1)}
~b={SinOsc.ar(300) * LFPulse.ar(1)}

// usamos dos canales con la ayuda de corchetes
~a = {SinOsc.ar([200, 400],0,0.5) * LFNoise0.ar([1,2])}
```

Ver ejemplo 10 en la carpeta de ejemplos.

Secuencias bajo demanda: Demand y Duty

Demand es un objeto que nos entrega valores de una lista bajo demanda de un *trigger,* el cual en SuperCollider se puede generar mediante cualquier señal que cambie su valor de no positivo a positivo.

La lista de valores se crea mediante un arreglo, mientras que la señal que puede usarse como *trigger* es Impulse.kr.

Para leer las listas o arreglos de valores en una secuencia utilizamos el objeto Dseq, que es un generador de secuencias bajo velocidad de demanda. En el caso de Dseq va

leyendo los valores del arreglo de principio a fin.

```
Demand.kr(Impulse.kr(1), 0, Dseq([0,1,2,3,4,5,6],inf))
```

En el ejemplo anterior podemos leer que Demand utiliza un Impulse.kr como *trigger,* el valor 1 indica el tiempo que el impulso va a cambiar de no positivo a positivo, luego el argumento 0 es el *reset* del Demand, y después viene el objeto que genera la secuencia, en este caso Dseq. Adentro de Dseq se encuentra una lista ordenada en un arreglo y su duración, en este caso inf, lo que indica que esa secuencia se va a repetir al infinito. En su lugar podríamos determinar la duración mediante el número de veces que queremos repetir la secuencia.

Hay diferentes tipos de generadores de secuencia que cambian el comportamiento en la lectura de los valores de la lista o arreglo, por ejemplo Drand tiene un comportamiento aleatorio.

```
// creamos una secuencia bajo demanda con lectura de valores aleatoria y la insertamos como control en el argumento de frec de un tono
~a={Demand.kr(Impulse.kr(8), 0, Drand([100,200,300,400,500,600],inf))}
~b={SinOsc.ar(~a.kr,0,0.1)}
~b.play
```

Otra forma de realizar secuencia bajo demanda es mediante el objeto Duty, la diferencia con Demand es que Duty tiene un Impulse incluído en su primer argumento, otra diferencia es que en el primer argumento es posible usar una secuencia bajo demanda como puede ser un Dseq, con lo cual podemos generar un comportamiento rítmico.

```
~d.play
~c={Duty.kr(1,0,Dseq([200,400,800],inf))}
~d={SinOsc.ar(~c.kr,0,0.5)}

// ejemplo con una secuencia bajo demanda en el primer argumento
```

```
~d.play
~c={Duty.kr(Dseq([1/4,1/2],inf),0,Dseq([200,400,800],inf))}
~d={SinOsc.ar(~c.kr,0,0.5)}
```

Ver ejemplo 11 en la carpeta de ejemplos.

Reloj y cuantización

Podemos referenciar las acciones realizadas dentro de un ambiente Proxy a un reloj y a su vez cuantizar dichas acciones, tener un reloj nos sirve para sincronizar los eventos que vamos programado y es útil cuando trabajamos con eventos rítmicos.

```
// establece un reloj basado en el tiempo inicial de arranque
p.clock = TempoClock.default

// indica que el próximo evento se va a ejecutar hasta la siguiente unidad temporal
p.quant = 1.0

// cambiamos la velocidad del reloj preestablecido, en tiempos por minuto
p.clock = TempoClock.default.tempo_(120/60)

// cambiamos la velocidad del reloj preestablecido, en tiempos por segundo
p.clock = TempoClock.default.tempo_(2.0)
```

El tiempo se puede especificar en tiempos por segundo, en cuyo caso usaremos números flotantes para indicarlo, o en tiempos por minuto en cuyo caso usaremos quebrados.

Ver ejemplo 12 en la carpeta de ejemplos.

Patrones

El tema de patrones es amplio, y aunque no es un tema exclusivo de JITLib, estos pueden usarse dentro del ambiente Proxy. Antes debemos explicar que *Patterns* es la forma en la que podemos programar mediante patrones en SuperCollider, de manera similar a programar un secuenciador. Los *Patterns* tienen una amplia variedad de estructuras que nos ayudan a componer de manera secuencial.

Dentro de ProxySpace un patrón se puede declarar de la misma forma que un sinte, el primer patrón que vamos a explicar es el Pbind cuya función es la de combinar valores de distinos tipos de patrones en un solo flujo de datos. Es importante mencionar que dentro de la estructura de *Patterns* ya existen argumentos preestablecidos, por ejemplo la duración se especifica \dur y la amplitud \amp.

```
~a.play
// este código hace sonar una nota de un sinte preestablecido cada medio
segundo
~a=Pbind(\dur, 0.5)
```

Después podemos agregar nuestro primer patrón anidado, que formará una secuencia de duraciones, para este fin usamos Pseq. Este patrón necesita de un array con elementos que indican los distintos tiempos de duración, los cuales serán leídos uno por uno de manera serial, después de nuestro array indicamos el número de veces que será ejecutado, si queremos que esto suceda de manera infinita usamos el mensaje inf.

```
// patrón con un arreglo de 5 elementos que se ejecuta de manera infinita
Pseq([1, 2, 3, 4, 5], inf)

// aplicado a nuestro ejemplo anterior
~a= Pbind(\dur, Pseq([0.25, 0.25, 0.5], inf))
```

```
// aplicamos Pseq a la duración y a la amplitud
~a= Pbind(\dur, Pseq([0.25, 0.25, 0.5], inf), \amp, Pseq([0.1, 1],inf))
```

Podemos agregar azar con el patrón Prand, el cual se estructura igual que Pseq solo que los valores no son leídos de manera secuencial sino de forma aleatoria.

```
// aplicamos Pseq a la duración y a la amplitud y Prand a frecuencia
~a=Pbind(\dur, Pseq([0.25, 0.25, 0.5], inf), \amp, Pseq([0.1, 1],inf),
\freq, Prand([200,400,800,1600],inf))
```

Ver ejemplo 13 en la carpeta de ejemplos.

Mensajes sobre patrones y valores midi

Métodos como .pyramid y .mirror agregan a los patrones un comportamiento en el que construyen cadenas de valores a partir de un modelo, por ejemplo mirror repite los valores del array en reversa una vez que estos terminaron y pyramid construye una una secuencia en forma de piramide.

Para que estos actuen sobre un patrón basta colocarlo después del array que esta dentro de un Pattern.

```
~a=Pbind(\dur, 0.5, \freq, Prand([200,400,800,1600].mirror,inf))
```

El mensaje .midicps nos permite trabajar con notas midi en lugar de valores de frecuencia. Recordemos que los valores de notas midi van de 0 a 127, donde 60 corresponde al do central.

```
~a=Pbind(\dur, 0.5, \freq, Prand([60,62,64,65,67,69,71].midicps,inf))
```

Trabajando con nuestros sintes: SynthDef

El tema de SynthDef es complejo, estos pueden usarse dentro de ProxySpace y es una manera más elaborada de trabajar sonidos. Mencionaré este tema pero sin profundizar en la construcción de ellos. El modelo de nuestro ejemplo puede ser utilizado para hacer variaciones de su estructura.

```
SynthDef(\sinte,{|out=0|
Out.ar(out,Pan2.ar(SinOsc.ar(200,0,0.1)*EnvGen.kr(Env.perc(0.1,0.1),doneAct
ion:2)))}).add
```

Limpiar el ambiente y salir de ProxySpace con .clear y .pop

Una vez que hemos terminado nuestra sesión de *live coding* o cuando alguna programación no resulto podemos limpiar el ambiente Proxy o una variable específica para lo que utilizaremos el método .clear, este método remueve los sintes, el monitor, el grupo y libera el bus.

Por último, en caso de querer salir de nuestro entorno de programación Proxy, basta con declarar .pop.

```
// limpia el contenido de la variable ~a
~a.clear
// limpia todo el contenido del ProxySpace
p.clear
// podemos especificar en cuanto tiempo realice esta acción
p.clear(8)
// salimos del ambiente Proxy
p.pop
```

Código y memoria: la clase History

La idea de conservar nuestra improvisación puede resultar interesante cuando usamos código, ya que además del resultado sonoro observar el proceso de programación paso a paso puede servirnos de referencia para analizar nuestra programación o reproducirla nuevamente.

Para realizar esta tarea podemos usar History, una clase programada por Alberto de Campo, la cual está diseñada para registrar el desarrollo de una programación a lo largo de una línea de tiempo, registrando cada cambio que realizamos sobre nuestra programación.

History comienza a generar un documento cuando le enviamos el mensaje .start y deja de generarlo con .end.

```
// comienza a registrar el historial
History.start

// cada vez que declaremos una linea de código, se registra una entrada en el documento
~a.play
~a={SinOsc.ar(200,0,0.1)}
~a.release

// paramos el registro
History.end
```

Para ver el resultado usamos .document, al ejecutar este mensaje se abre una nueva venta que nos muestra la historia.

```
History.document
```

El documento resultante nos indica el tiempo en que cada nueva línea de código fue declarada. Con .play podemos reproducir nuestra ejecución y con .stop dejar de reproducirla.

```
History.play
History.stop
```

Para conservar nuestra historia de cada programación es necesario salvarlas, para lo cual utilizamos .saveStory, indicando una ruta y asignando un nombre. La ruta se indica entre parentesis y comillas y el nombre del documento lleva la extensión .scd

```
History.saveStory("/Users/compu/Documentos/historia1.scd")
```

Para llamar una historia de programación anterior utilizamos .loadStory, indicando entre parentesis y comillas, la ruta donde guardamos el documento.

```
History.loadStory("/Users/compu/Documentos/historia1.scd")
```

Para comenzar una nueva historia, podemos borrar los datos que se han quedado guardados en el documento del historial con el mensaje .clear, de otra manera los nuevos datos comenzarán a guardarse después del registro de la última programación.

```
History.clear
```

Una función útil de History es la ventana de visualización, con .makeWin se crea esta ventana donde podemos ver como se va desarrollando nuestra programación en el formato de History. Esta ventana contiene varias botones que nos permiten realizar algunas acciones desde la ventana como star/end.

```
History.makeWin
```

Ver ejemplo 14 en la carpeta de ejemplos.

Live Coding colectivo y redes

La práctica de *live coding* también puede realizarse de manera colectiva, al haber varios participantes y computadoras involucradas en una improvisación, esta podría llevarse a cabo siguiendo distintas estrategias, una primera aproximación es que cada participante del ensamble produzca sonidos de manera individual sin sincronía y sin compartir datos.

Otra estrategia es poder enviar datos y sincronizar los relojes de las computadoras involucradas para lo cual se establece una red para enviar mensajes bajo el protocolo OSC.

Primero es necesario saber la dirección IP[7] de nuestra computadora la cual nos identifica dentro de una red y está compuesta por 4 números divididos por puntos, por ejemplo la dirección interna de una computadora es “127.0.0.1”.

Una vez identificada la dirección IP de nuestra computadora, necesitamos saber como mandar y recibir mensajes OSC entre distintas computadoras. En SuperCollider se usa `NetAddr` para mandar los mensajes y `OSCresponder` para recibirlos. Los mensajes contienen una etiqueta que los identifica y un valor del mensaje, aunque estos no llegan solos, vienen acompañados de otra información. Para esta primera aproximación lo que interesa es un valor que podamos utilizar para modificar algún argumento en otra computadora.

```
// enviar datos de manera interna al puerto  de SuperCollider que es 57120
n=NetAddr("127.0.0.1", 57120)
// recibir datos e imprimir solo el valor
o=OSCresponder(nil, "/mensaje", {|...msg|
msg[2][1].postln;
}).add

// el mensaje a enviar se compone por la etiqueta y el valor
n.sendMsg("/mensaje", 500)
```

7 Internet Protocol

Ver ejemplo 15 en la carpeta de ejemplos.

Otros programas para Live Coding

Por último resta mencionar que, la librería JITLib es una herramienta muy potente y flexible para la práctica de *live coding*, en SuperCollider también se puede programar fuera del ambiente Proxy mediante el uso de Pdef, Tdef y Ndef.

Asimismo, además de SuperCollider otros programas como ChucK o Extempore son usados para programar sonido. En el caso de la programación de imágenes Fluxus o Processing son de los programas más utilizados. Bajo el paradigma de *live patching* Pure Data y Max/MSP. Incluso el lenguaje C aplicado a microcontroladores.

Bibliografía

Collins, N., McLean, A., Rohurhuber, J., y Ward, A. (2003). *Live coding in laptop performance*. Cambridge: Cambridge University Press.

De Campo, A., Rohrhuber, J. y Weiser, R. (2004). *Algorithms today: Notes on language design for just in time programming*.

Dodge, Ch. y A. Jerse, T. (1997). *Computer Music: Synthesis, composition and performance*. Schirmer.

McLean A. (2011). *Artists-Programmers and Programming Languages for the Arts.* (Tesis doctoral). Goldsmiths University of London, Londres.

Noble, J. (2009). *Programming Interactivity: A designer's Guide to Proccesing, Arduino and openFrameworks*. Sebastopol: O'Reilly Media.

Romero, E. y Villaseñor, H. (2012). *Música por computadora*. México, DF: Centro Multimedia.

Rohrhuber, J., De Campo, A. y Olofsson, F. (2002). *Just in Time Programming Library Documentation*. SuperCollider Distribution 2002-2013.

Sorensen, A. y R. Brown, A. (2007). *aa-cell in practice: an approach to musical live coding*. International Computer Music Conference, Copenhague.

Wilson, S., Cottle, D. y Collins, N. (ed.). (2012). *The SuperCollider Book*. Massachusetts:The MIT Press.