

Live Coding

El paradigma de la programación en vivo

Hernani Villaseñor Ramírez

Centro Multimedia, México, D.F. 2012

¿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.

La práctica de programar en vivo también se conoce como *on-the-flying-programming*, *conversational programming* o *interactive programming*.

***Live Coding* en México: una perspectiva desde el Centro Multimedia**

La práctica de *live coding* en el Centro Multimedia tiene sus orígenes en un concierto de la agrupación Mu, la cual estaba formada por Eduardo Meléndez, Ezequiel Netri y Ernesto Romero quienes trabajaban en el Taller de Audio de este centro. Esta presentación tuvo lugar en Radio Ibero. El 14 de julio 2009, primer concierto de Sinescencia¹, Ezequiel Netri presentó

¹ Sinescencia 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.

“Prácticas con código en vivo: Re-modulación en vivo de código abierto a través de Supercollider”, que junto al concierto de Mu podrían considerarse las primeras presentaciones de *live coding* en el CMM.

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 a cabo en la sala Manuel Felguérez del CMM. Y es así que a partir del 2011, el Taller de Audio del CMM organiza mensualmente una sesión de *live coding*².

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, Arduino, lenguaje c y en ocasiones 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, y cada sesión comenzó a organizarse en una sede distinta, al CMM, atrayendo así, nuevos públicos. Lugares como SAE Institute, Fonoteca Nacional, Ciencias UNAM y el Laboratorio de Video en la Esmeralda han sido nuevos escenarios para este esfuerzo de difusión, que poco a poco se ha 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

Las sesiones de *live coding* en el CMM, además de servir como plataforma para difundir una práctica y crear una comunidad de artistas/programadores, ha sido detonador para la organización de otros eventos ligados a esta práctica, por ejemplo Diarios Efímeros es un ciclo organizado por Jaime Lobato dentro de IndexMUAC, programa de música

² Actualmente se han realizado 20 sesiones de Live Coding en el CMM, promovidas por el Taller de Audio.

contemporánea 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*³.

Una parte importante en las prácticas artísticas emergentes es la reflexión, así que, de esta necesidad surge el Simposio Internacional de Música y Código */*vivo**, el cual tendrá lugar en el CMM del 13 al 16 de noviembre 2012 con el tema: Live Coding.

Como figuras destacadas de esta comunidad de artistas/programadores en torno a las sesiones de Live Coding del CMM se puede mencionar a Ernesto Romero que, junto a quien escribe este documento, han sido organizadores y promotores de la programación en vivo en el este espacio. También podemos mencionar a Luis Navarro, Eduardo Obieta, Martín Zumaya, Jaime Lobato, Alejandro Franco, Mitzi Olvera, Alexandra Cárdenas, Emilio Ocelotl, Jaime Jalil y José Carlos Hasbun entre otros.

Introducción a SuperCollider

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

³ Modificación de juguetes de pila.

sistema operativo que usamos y por la forma en la que corremos SuperCollider¹. 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:

sistema	encender	apagar
Mac	enter	cmd + punto
Windows	ctrl + intro	alt + punto
Gedit (Ubuntu)	ctrl + e	esc

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

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

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 parecerse 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 servidor 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 está 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

Las 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 están 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 anteceditas 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 cuántos 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 sinusoidales: 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 comportamiento 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 Rohrer. JITLib significa “*Just in time programming*” otra forma de referirse 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 antepone 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 sint con los mensajes play/stop, el
sinte se libera del server
~a.play
~a={Sin0sc.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 con pause/resume, el sinte no se
librea del server
~a.play
~a={Sin0sc.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 con release/send, el sinte se libera
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)
```

```
// en el caso de nuestra variable ~salida, cada cambio responderá al tiempo
asignado solo a esa 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={lfreq = 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*⁴ 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

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.

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

4 Cruce de señales mediante una transición de tiempo.

```
// 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={lfrec=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.

Secuencias con Demand

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 crea mediante un arreglo y una 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` va a utilizar un `Impulse` 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 freq 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)}
```

Aleatoridad y reconstrucción de valores

Hay varios métodos para crear valores aleatorios en SuperCollider, por ejemplo `rrand` o `exrand`, 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.

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 11 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 distintos 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 12 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.

```
~sinte=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
```

Bibliografía

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*. Goldsmiths University of London: Londres.

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

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

Ayuda de JITLib

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



Esta obra está sujeta a la licencia Reconocimiento-NoComercial-CompartirIgual 3.0 Unported de Creative Commons. Para ver una copia de esta licencia, visite <http://creativecommons.org/licenses/by-nc-sa/3.0/>.