Música por computadora

Ernesto Romero y Hernani Villaseñor

Centro Multimedia 2012

Sesión 4

4 Herramientas propias de SuperCollider: ProxySpace, Demand

ProxySpace

Es parte de la librería de programación en tiempo real: JITLib o Just In Time Programming – Library.

Esta librería fue desarrollada por Julian Rohrhuber con la ayuda de Alberto de Campo y Fredrik Olofsson, la cual muestra los conceptos básicos de programación interactiva con SuperCollider y *ProxySpace.*

¿Programar en vivo?

Gracias a la posibilidad de declarar código sin tener que frenar las tareas que realiza el programa, la programación en vivo se hace posible. En el caso de SC la librería JITLib nos ayuda a realizar está práctica de manera muy efectiva, aunque no es la única técnica en la que podemos modificar código en tiempo real.

¿Qué es ProxySpace?

Un Proxy es un parámetro de sustitución, para algo que está sucediendo en el Servidor. Este parámetro puede ser usado para operar en algo que aún no existe. Cualquier objeto puede tener un comportamiento Proxy, especialmente las funciones.

Como iniciar ProxySpace

Para iniciar un ambiente Proxy o ProxySpace comenzamos declarando el siguiente código, que empuja la entrada a la librería JITLib, la cual nos permite modificar código en el vuelo para realizar prácticas como la de Live Coding.

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

~out.play
```

Programamos a partir de funciones, las cuales contienen objetos que funcionan dentro de la lógica del Proxy, es decir que sus argumentos pueden ser modificados en el momento.

```
~out={SinOsc.ar(440,0,1)}
```

Argumentos y .set

Implementamos un argumento el cual podremos cambiar mediante el método .set, podemos asignar cuantos argumentos queramos y darles cualquier nombre siempre que empiecen con minúscula. Para usar el método .set se engloba entre paréntesis los argumentos que se van a modificar, se agrega una diagonal al nombre del argumento y a continuación se indica el nuevo valor, todo se separa por comas.

```
//creamos el argumento frec, lo igualamos a 440 y lo sustituimos
dentro de los argumentos de SinOsc

~out={|frec=440| SinOsc.ar(frec,0,1)}

//con el mensaje .set cambiamos la frecuencia
~out.set(\frec, 880)
```

Nota como cambia inmediatamente la frecuencia, sin necesidad de parar el código y volverlo a declarar, esta es la flexibilidad y poder de la programación en tin tiempo real. La sintáxis es importante, si te fijas el argumento que delclaramos como frec, al momento de implementarlo con el método .set se le añadie antes una diagonal, \frec.

Veamos un ejemplo de como implementar dos argumentos y modificarlos.

```
//creamos el argumento amp, lo igualamos a 1 y lo sustituimos
dentro de los argumentos de SinOsc

~out={|frec=440, amp0=1| SinOsc.ar(frec,0,amp)}

//con el mensaje .set cambiamos la frecuencia y la amplitud
~out.set(\frec, 880, \amp, 0.5)
```

.fadeTime y .xset

fadeTime es un método que implementa un desvanecimiento gradual de volumen cada ves que modificamos un argumento o que ponemos en pausa el Proxy. El método .xset funciona igual que .set, la diferencia es que .xset genera un cruce de volumenes (*crossfade*) entre el valor anterior del argumento y el nuevo valor. El tiempo que dura el cruce de volumenes está determinado por al valor que asignamos a fadeTime.

```
~out={|frec=440, amp0=1| SinOsc.ar(frec,0,amp)}

// usamos .fadeTime para asignar un tiempo de desvanecimiento en segundos
~out.fadeTime=5

//con el mensaje .xset cambiamos la frecuencia y veamos como el valor anterior y el nuevo se cruzan en un tiempo de 5 segundos
~out.xset(\frec, 880)
```

Canales

Al utilizar ProxySpace de esta manera la salida del sonido es por el primer canal de SuperCollider, o sea el lado izquierdo, para salir en ambos canales podemos usar el UGen Pan2.ar

```
//salida solo por el canal izquierdo
~out={Pulse.ar(2,0.5,0.1)}

// salida por ambos cnalaes, usando una iteración !2
~out={Pulse.ar([1,2],0.5,0.1)}

// salida por ambos canales usando Pan2.ar
~out={Pan2.ar(Pulse.ar(2,0.5,0.1),0,1)}
```

pause/resume

Con el mensaje .pause mandado a la variable ponemos en pausa las funciones o las operaciones asignadas a ella. Con resume regresamos de la pausa lo que esta contenido en la variable. El sintetizador sigue activo mientras está en pausa.

```
~out.pause
~out.resume
```

send/release

Send y release son mensajes que parece que hacen la misma funcion que los anteriores pause/resume, la diferencia es que send enciende un nuevo sinte y release lo libera. El efecto auditivo es que entra y sale suavemente el sonido dependiendo del tiempo de desvaneciemiento.

```
~out.release
```

```
~out.send
```

rebuild y el factor aleatorio

Con los mensajaes rand y exprand podemos agregar comportamiento aleatorio a los UGens. El mensaje .rebuild nos reconstruye los valores *random* pudiendo así modificar los valores de argumentos en random con tan solo mandar el mensaje desde la variable.

```
// un sinte con valores aleatorios, en este caso rrand es aplicado a el argumento de frecuencia

~out={SinOsc.ar(rrand(100,1000),0,0.1)}

// declarando esta línea de código reconstruye los valores random

~out.rebuild
```

p.clear

Remueve todos los sintes, grupos y monitor además libera el bus, el número entre paréntesis indica el tiempo en segundos que tardará en realizarse esta acción.

```
p.clear(10)
```

p.pop

Este mensaje es usado para salir del ambiente Proxy.

```
p.pop
```

Demand

Demand.kr (trigger, reset, demand rate Ugens)

Extrae elementos de uno o varios arrays con cierta periodicidad y con cierto orden.

La clase Demand trabaja siempre junto con otra clase de tipo *demand rate*. Estos UGens se escriben siempre empezando con la letra D mayúscula. Por ejemplo Dbrown, Dgeom, Dseries, etc. Los *demand rate* UGens definen los elementos de un arreglo y la manera en que serán entregados cuando el Demand lo solicite. Esta entrega puede ser de varias formas: en secuencia, aleatoriamente, siguiendo un patrón aritmético o geométrico, etc. Demand puede trabajar con .kr o con .ar cuidando que el trigger si es un UGen sea del mismo rate (e.g. Demand.ar (Impulse.ar).

Los argumentos de Demand son:

Trigger: Puede ser una señal o un argumento que se modifica desde afuera. Cuando el trigger cambia de no positivo a positivo se extrae un elemento.

Reset: Reinicia los Demand Ugens de modo que el siguiente elemento que se extraiga sea el primero de cada UGen.

Demand UGens: Puede ser cualquiera de los *demand rate* UGens como Dseq o Drand. Estos Ugens contienen los arreglos de donde el Demand extrae los elementos. Cada UGen se comporta de manera distinta el entregar los elementos de los arreglos.

Veamos algunos *demand rate* UGens:

Dseq (array, length)

Entrega en orden los elementos de un array un determinado número de veces.

array: array con los elementos en orden que se quieren entregar al Demand.

length: número de repeticiones. Una repetición es la lectura de todo el array.

Ejemplo 1

Aqui creamos dos argumentos: t_trig para pedir un elemento nuevo del array y t_reset para reiniciar el Dseq. Es necesario usar este tipo de argumentos antecedidos por t_ para que funcione. En el tercer argumento del Demand ponemos un Dseq que nos entregará en secuencia los elementos de su array infinitas veces. El Demand está asignado a la variable ~freq que hace las veces de frecuencia para un SinOsc.

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

~out.play

~freq={|t_trig,t_reset|
Demand.kr(t_trig,t_reset,Dseq([440,493.88,554.36,587.32,659.25,7
39.98,830.6,880],inf))};

~sig={SinOsc.ar(~freq.kr)};

~freq.set(\t_trig, 1)
~freq.set(\t_reset, 1)

~out=~sig
```

Ejemplo 2

Ahora usamos una señal como trigger. La señal es un Impulse que pedirá un elemento al Dseq 2 veces por segundo. Dseq entregará los elementos de su array una sola vez.

```
~freq={|t_reset|
Demand.kr(Impulse.kr(2),t_reset,Dseq([440,493.88,554.36,587.32,6
59.25,739.98,830.6,880],1))};
~sig={SinOsc.ar(~freq.kr)};

~freq.set(\t_reset, 1) // vuelve a empezar la secuencia desde el
primcipio

~out=~sig

~out=0
```

Drand (array, length)

Entrega en orden aleatorio los elementos de un array un determinado número de veces.

array: array con los elementos que se quieren entregar al Demand.

length: número de elementos del array que se entregarán. Diferente de Dseq donde se cuentan las veces que se entrega todos los elementos. En Drand se cuenta elemento por elemento.

Ejemplo 3

Igual que el Ejemplo 1 pero con Drand. Aqui el uso del t_reset es irrelevante ya que, al ser aleatorio el orden, no hay un primer elemento.

```
~freq={|t_trig|
Demand.kr(t_trig,0,Drand([440,493.88,554.36,587.32,659.25,739.98
,830.6,880],inf))};

~sig={SinOsc.ar(~freq.kr)};

~freq.set(\t_trig, 1)

~out=~sig
```

Ejemplo 4

Ahora usamos una señal como trigger. La señal es un Impulse que pedirá un elemento al Drand 2 veces por segundo. Dseq entregará solo 4 elementos de su array.

```
~freq={Demand.kr(Impulse.kr(2),0,Dseq([440,493.88,554.36,587.32,
659.25,739.98,830.6,880],4))};

~sig={SinOsc.ar(~freq.kr)};

~out=~sig

~freq.rebuild  //  para que vuelva a arrojar 4 elementos al azar

~out=0
```

Ejemplo 5

Se pueden poner otros demand rate Ugens como elementos del array de un demand rate Ugen

```
~freq={Demand.kr(Impulse.kr(8),0,Dseq([Dseq([211.8,200,211.8,224
.4,211.8,200,211.8],1), Drand
     ([399.9,336.3,158.7,118.9],1)],inf))};
 ~sig={Blip.ar(freq*200, 2)};
```

Ejemplo 6

```
~freq={Demand.kr(Impulse.kr(8),0,Dseq([Dseq([211.8,200,211.8,224
.4,211.8,200,211.8],1), Drand
     ([399.9,336.3,158.7,118.9],1)],inf))};
~armonicos={Demand.kr(Impulse.kr(10),0,Dseq([4,2,3,4,5,6,7,8,9,1
0,11,12,13].pyramid,inf))};
~sig={Blip.ar(~freq.kr, ~armonicos.kr)};

// ejemplo del help de Demand

(
{
     var trig, seq;
     trig = Impulse.kr(12);
     seq = Drand([
     Dseq([4,1,0,2,2,0,1,2]),
     Dseq([1,0,0,0,2,1]),
     Dseq([4,1,0,1,1]),
     Dseq([4,4,0,0])], inf);
     trig = Demand.kr(trig, 0, seq * 0.4) * trig;
     {LPF.ar(PinkNoise.ar, 12000)}.dup * Decay.kr(trig, 0.5);
}.play;
)
```

Dswitch (array, index)

Posee un array de donde puede escoger a voluntad un elemento y mandarlo al Demand.

array: array con los elementos que se pueden entregar al Demand. Pueden ser números, demand rate Ugens o alguna otra Clase.

index: Indice del elemento del array que queremos entregar al Demand.

Ejemplo 6

Usamos un Dswitch para escoger diferentes patrones numéricos y asignarlos a la frecuencia de un Pulse. Creamos el argumento switch dentro de la funcion para poder escoger el índice del array del Dswitch.

```
~freq={|switch| Demand.kr(Impulse.kr(8), 0,
Dswitch1([Dseq([300,336.7,377.9], inf), Dseq([300,336.7,356.7],
inf), 300, MouseY.kr(600,300)], switch))};
~sig={Pulse.ar(~freq.kr)!2};

~out=~sig

~freq.set(\switch, 0)  //  Dseq([300,336.7,377.9], inf)
~freq.set(\switch, 1)  //  Dseq([300,336.7,356.7], inf),
~freq.set(\switch, 2)  //  300
~freq.set(\switch, 3)  //  MouseY.kr(600,300)
```

Ejemplo 7

Se puede utilizar el Demand con audio rate para generar ondas. Recuerden que el trigger debe tener audio rate también

```
Array.series(100,0,0.1).sin.plot  //  los puntos que forman la
onda

~sig={|freq=7033|Demand.ar(Impulse.ar(freq), 0,
Dseq(Array.series(600,0,0.1).sin,inf))};

~out=~sig

~sig.set(\freq, rrand(200,7033))

s.scope

~sig={|freq=7033| Demand.ar(Impulse.ar(freq),0,Dseq([0,-
1,0.4,0.6,0.9,0.1,1,0.5],inf))}

~sig.set(\freq, rrand(200,7033))

~sig={|freq=7033|
Demand.ar(Impulse.ar(freq),0,Dseq(Array.geom(10,1,0.28).reverse.
mirror,inf))}
```