A dynamic spatial locator ugen for CLM

Fernando Lopez-Lezcano*
nando@ccrma.stanford.edu
*CCRMA, Stanford University

Abstract— **Dlocsig is a dynamic spatial locator unit generator written for the Common Lisp Music (CLM) sound synthesis and processing language. Dlocsig was first created in 1992 as a four channel 2d dynamic locator and since then it has evolved to a full 3d system for an arbitrary number of speakers that can render moving sound objects through amplitude panning (VBAP) or Ambisonics. This paper describes the motivations for the project, its evolution over time and the details of its software implementation and user interface.**

I. Introduction

CLM is a very powerful synthesis and sound processing language in the style of Music N languages written originally in 1989 in Common Lisp by Bill Schottstaedt[2] (it was optimized at that time for running on the NeXT computer and its built in Motorola DSP). I started using it for music composition at the end of 1990, shortly after its creation.

Regretfully the NeXT workstation only had CD quality stereo outputs, which was a "downgrade" from the SamsonBox[12] four channel output, so the original version of *dlocsig* was created for the QuadBox, an external four channel D/A converter connected to the DSP port of the NeXT. The original QuadBox hardware was designed by Atau Tanaka at CCRMA and the firmware and playback software for the NeXT was programmed by myself while working at Keio University in Japan in 1992 - an "across the Pacific" joint project made possible by the Internet.

CLM[1] included "locsig", a simple panning based ugen for stereo signal location. It fell short of my needs, so I started writing another unit generator ("dlocsig", **d**ynamic **locsig**) that would encapsulate all the behavior needed to simulate most spatial cues of moving sound objects, and most importantly would be a drop in replacement for locsig so that it would be easy to modify existing instruments to use it.

Work in *dlocsig* started in 1992 and continues to this day, and the unit generator has been used by myself and other composers in numerous pieces. The original version was a four channel two dimensional system and used pair-wise panning between adjacent speakers[3]. An Ambisonics rendering back-end was added in 1999 (for B-format output and pre-rendered Ambisonics for known speaker configurations). In 2000 the ugen was extended to cover 3d arbitrary arrangements of speakers and included 3d Ambisonics and VBAP for amplitude panning. It has been part of the CLM distribution for a long time now (I don't remember when it was originally incorporated). In 2006 Bill Schottstaedt changed the name of the basic unit generator to move-sound while providing backwards compatibility synonyms within CLM.

CLM currently supports Scheme, Ruby and Forth languages in addition to the original Common Lisp, and *dlocsig* has been ported to the Scheme and Ruby worlds. The software is GPL and all source is available as part of the CLM distribution.

II. The unit generator

The current unit generator can generate spatial positioning cues for any number of speakers which can be arbitrarily arranged in 2d or 3d space. The appropriate speaker configuration is selected based on the current number of channels in the output stream. In pieces which can be recompiled from scratch this feature allows the composer to easily create several renditions of the same piece, each one optimized for a particular listening environment and rendering technique. Each user-defined speaker arrangement can also include delay compensation for the speakers and can map each speaker to an arbitrary channel in the rendered output stream.

The unit generator has several back-ends for rendering the output sound file with different techniques. The default is amplitude panning between adjacent speakers (between adjacent speakers in 2d space or three speaker triads in 3d space using VBAP[4]). It can also create an Ambisonics[5] first order b-format four channel output sound file suitable for feeding into an appropriate decoder for multiple speaker reproduction. Or it can decode the Ambisonics encoded information on-the-fly to an arbitrary number of output channels if the speaker configuration is known in advance.

An additional back end that can render 3d movements over stereo speakers or headphones using HRTFs was designed and coded in 2001 but was never finished or released.

All existing rendering back-ends can be combined while rendering a piece, and all of them take care of corner conditions like diagonal paths that cross 0,0,0 by appropriately changing the rendering details. In all cases standard cues like Doppler, multichannel output to a reverberator, amplitude scaling due to distance for the direct and reverberated components of the sound (with user-defined exponents) and ratio of direct to reverberated sound are also automatically generated.

A. Implementation

Like the rest of CLM, the original *dlocsig* core was written in Common Lisp. It is actually a Common Lisp macro that generates Lisp code on the fly to be inserted

into the run loop of the instrument (the "run loop" is the section of a CLM instrument that generates the samples).

CLM unit generators are usually created and used through two functions or macros. One creates the unit generator data structures and is executed at the beginning of the instrument run, the other is usually a macro that executes the ugen code for each sample to be generated and is connected to other ugens through arbitrary lisp code.

For efficiency reasons the bulk of the complexity of the ugen was shifted to make-dlocsig, the ugen creation function. Its output is a list of parameters which the ugen itself uses to render the localized samples. The ugen itself does not know anything about rendering methods or trajectories in space and currently only knows how to apply individual amplitude envelopes to each of the output channels.

This is the Lisp definition of make-dlocsig and all its parameters (default values and some additional parameters omitted for brevity):

```
(defun make-dlocsig (start-time duration
   path
   scaler
   direct-power inside-direct-power
   reverb-power inside-reverb-power
   reverb-amount
   initial-delay
   unity-gain-dist
   inside-radius
   minimum-segment-length
   render-using)
```

start-time and *duration* define the start and duration of the sound being rendered, *path* is a path object (see below), **-power* arguments can be used to control the power exponent for attenuation due to distance for both the direct signal and the reverberated signal, and *render-using* defines the type of rendering to be done (VBAP amplitude panning, Ambisonics, etc). *inside-radius* defines the diameter of the sphere where limiting of the output signal amplitude is done and *unity-gain-dist* defines the distance at which unity gain scaling is done for the input signal.

A call to make-dlocsig returns a list that contains all the information needed by the unit generator to render the sound and the values for the start and end indexes of the enclosing run loop, as its start and duration can be affected by radial velocity and the Doppler effect (if the initial and final distances of the moving object differ).

The list contains (amongst other components) gain arrays for the direct and reverb signals with individual envelopes defined for each output channel based on the movement and the rendering method selected. Because of this internal rendering to amplitude envelopes, the same unit generator can render both amplitude panning and Ambisonics without any changes. The structure also contains an envelope for the radial velocity component of the movement so that Doppler can be generated through the use of an interpolated delay line in the unit generator.

B. Reverberation

Reverberation is not integrated into the dlocisg unit generator. It uses the standard CLM conventions for reverberation unit generators. The first versions of *dlocsig* used a tweaked four channel version of nrev, one of the most popular reverbs in the times of the Samson Box. The current version of *dlocsig* comes bundled with an n-channel version of freeverb which can reverberate n-channel inputs to n-channel outputs with a choice of local reverb percentage in the case of multichannel inputs.

C. Global Configuration

Several configuration variables can be set to globally alter the behavior of *dlocsig*.

- *dlocsig-one-turn*: the number that represents one turn, defines the angle units to be used
- *dlocsig-speed-of-sound*: defines the units of measurement for distance through the speed of sound
- *dlocsig-3d*: defines whether 2d or 3d speaker configurations are used by default
- *path-3d*: defines how paths are parsed when submitted as a plain list

In addition each of the parameters of make-dlocsig have default values based on global variables.

D. Speaker Array Configuration

Dlocsig can render soundfile output to any number of speakers when rendering to VBAP amplitude panning or to "rendered Ambisonics" (the output format for Ambisonics rendering is obviously independent of the speaker configuration).

The function *arrange-speakers* can be used to create speaker configurations for 2d or 3d setups. The speaker position is defined by their angles with respect to the listener. An additional delay can be specified for each of them in terms of time or distance, and speakers can be mapped to arbitrary output channels. Here is the definition:

```
(defun arrange-speakers (
   (speakers '())
   (groups '())
   (delays '())
   (distances '())
   (map '()))
```

speakers is a list of speaker positions in space defined using azimuth and elevation angles. Indexes from 0 in this list are used to explicitly define groups of speakers in 2D or 3D space (using the *groups* parameter - each group is a panning group of related speakers). *delays* and *distances* can be used to add delay compensation for individual speakers, and finally *map* can map any speaker to any output channel to generate custom output soundfiles that are adapted to a particular mapping of speakers in the final delivery system.

Dlocsig pre-defines a number of "reasonable" configurations for standard setups. Predefined configurations are

indexed by number of output channels. A global variable (*dlocsig-3d*) is used to differentiate between flat 2D and 3D speaker arrangements. The number of output channels and the global variable are used to select a configuration at runtime and all the rendering is adjusted accordingly.

III. The Path Object

The movement of sound sources in space is described through path objects. They hold the information needed by the unit generator to move the source in space and are independent of the unit generator itself and the rendering technique used (ie: the composer uses a front end that is independent of the rendering technique used and number of output channels). Path objects can be reused and can be translated, scaled and rotated in 3d space as needed. There are several ways to describe a path. Bezier paths are described by a set of discrete points in 2d or 3d space that are joined by bezier segments through curve fitting. This description is very compact and easy to specify as a few points can describe a complex trajectory. Paths can also be specified in term of geometric entities (spirals, circles, etc). A user-defined function can also generate the points and incorporate them into a path object. In all path descriptions the velocity profile of the movement can also be specified as a function of distance.

A. Bezier Paths

This is the generic path creation function for paths defined through discrete points in space:

```
(defun make-path (path
   (3d path-3d)
   (polar nil)
   (closed nil)
   (curvature nil)
   (error 0.01)
   ;; only for open paths
   initial-direction
   final-direction)
```

The first argument, *path* is a list that specified the coordinates of the path the sound object will follow in space. Each component of the list is a list which describes a point in space and an optional relative velocity. The coordinates of each point can be specified in terms of cartesian coordinates (x, y, z) or polar coordinates (azimuth, elevation and distance - if the *polar* argument is non-nil). Paths can be open or *closed* (in the later case the initial and final points have to match). If a path is open both *initial-direction* and *final-direction* can be specified and will define direction vectors for the start and end of the movement.

If a velocity profile is not specified, the moving virtual object starts and ends at rest. The velocity profile is translated into absolute velocities for each segment of the movement by using a "constant acceleration" paradigm. Velocity is continuous at the segment boundaries and acceleration changes in a step function, being constant within each segment.

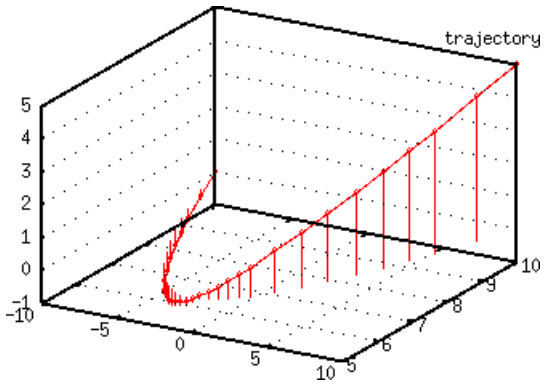

Fig. 1. Trajectory of sound object.

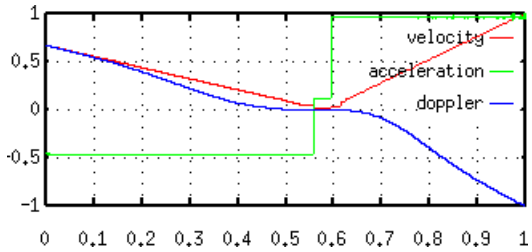

Fig. 2. Trajectory of sound object.

Here is the code that creates a very simple path expressed in cartesian coordinates:

```
(make-path
 '((-10 10 0 1)(0 5 0 0)(10 10 5 1.5)))
```

And the corresponding 3d plot of trajectory (fig. 1) and velocity, acceleration and Doppler frequency shift (fig. 2)

The path is internally rendered using a bezier curve approximation to the supplied coordinates. Each supplied point becomes a control point in the bezier curve and the control vectors are automatically derived using a curve fitting algorithm. For efficiency reasons (which may not be valid today) the rendering of the bezier curve is not done in the unit generator itself, the curves are pre-rendered to individual piece-wise linear envelopes for each output channel in the process of creating the ugen. As such, the bezier curve is approximated by individual straight line segments that are very cheap to render at sample generation time and are close enough to the original bezier curve that the Doppler shift artifacts due to the sudden change of direction at each inflection point are inaudible. The precision of the rendering process can be controlled through the *error* parameter, which defines the error bound of the linear segment approximation. The *curvature* argument controls the length of the control point vectors of the bezier curve segments so that the curvature of the bends at each control point can be controlled (see fig. 3 and fig. 4).

B. Geometric Paths

Some path subclasses exist that make the generation of some very common paths easy, in particular paths related to geometric shapes.

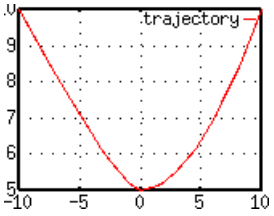

Fig. 3. :curvature '(0.4 1)

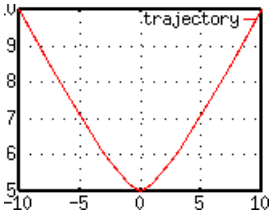

Fig. 4. :curvature 0.4

```
(defun make-spiral-path ((start-angle 0)
    total-angle
    (step-angle (/ dlocsig-one-turn 100))
    turns
    (distance '(0 10 1 10))
    (height '(0 0 1 0))
    (velocity '(0 1 1 1)))
```

Arguments should be obvious. It is possible to create arbitrary spirals from envelope like descriptions of the starting angle, total angle or number of turns and distance, with height and velocity profiles.

C. Literal Paths

Another class can be used to pack specific points into a path object without any further rendering or approximation done on the points in space. This makes it easy to use functions to create paths of arbitrary complexity.

D. Path Transformations

Path objects can be modified with some predefined transformations. They can be scaled, translated along all three axis and rotated along an arbitrary rotation point and direction vector. Paths can also be mirrored along a mirror vector. All these transformations do not affect the original coordinates of the path object, which can be reset to its original state at any time. In this way it is possible to define a set of paths and then transform them into families of paths that are used to define the movement of related sound objects in space.

E. Path Visualization

The path objects are created from text representations and not through a graphical editor. But methods are provided that can be used in conjunction with external visualization programs (gnuplot) to visualize the final rendition of the path as defined in the object.

It is possible to plot the trajectory, velocity profile, acceleration profile and Doppler shift of the moving sound object.

F. Examples

A very simple Lisp instrument that uses dlocsig:

```
(definstrument sinewave (start-time duration freq amp
    &key
    (amp-env '(0 1 1 1))
    (path (make-path
             :path '(-10 10 0 5 10 10))))
  (multiple-value-bind (dloc beg end)
      (make-dlocsig :start-time start-time
                    :duration duration
                    :path path)
    (let* ((osc (make-oscil :frequency freq))
           (aenv (make-env :envelope amp-env
                           :scaler amp)))
      (run
       (loop for i from beg below end do
         (dlocsig dloc i (* (env aenv)(oscil osc))))))))
```

This snippet of code will render one note created with the previously defined instrument in a four channel two dimensional setup:

```
(with-sound(:channels 4)
  (sinetest 0 1 440 0.5 :path
      (make-path '((-10 10)(0 5)(10 10)))))
```

The same instrument could render a 3d path in a cube of 8 speakers:

```
;; tell the system I want to use 3d
(setf dlocsig-3d t)
;; render the sound with a 3d path
(with-sound(:channels 8)
  (sinetest 0 1 440 0.5 :path
      (make-path '((-10 10 0)(0 5 10)(10 10 5)))))
```

IV. Limitations

The use of an interpolated delay line to render the Doppler frequency shift imposes a limitation to CLM instruments that use it. CLM is by nature a "sound painting" environment. It does not require notes to be time ordered in its score (which is just a Lisp program), not does it require samples to be output in time order from within an instrument (ie: any instrument can sprinkle sounds at arbitrary times in the output sound file). That absolute freedom in the time ordering domain does not mesh with a delay line that has to be fed with a constant stream of samples, so some instruments are not compatible with *dlocsig*.

An example is grani, a general purpose granular synthesis instrument I started writing in 1996. Generated grains are not necessarily time ordered and thus cannot be fed to *dlocsig* inside the instrument itself.

In these cases it is relatively easy to work around the problem by using *sound-let* and a very simple instrument that can move an arbitrary sound file in space (it is provided as an example in the CLM distribution). Sound-let calls the troublesome instrument and creates an intermediate sound file which is later spatialized by *dlocsig*. The process is transparent to the composer and the additional time overhead of the intermediate sound file creation is not significant.

V. Future Directions

There are many things in my list of "things to do" for Dlocsig, here are some details about the most important of them:

- The Ambisonics encoding back end is being expanded to include second (and higher) order Ambisonics encoding[6] [7].
- The Ambisonics rendering back end (used when the selected rendering type is *decoded-ambisonics*) is too simple, it should be extended to do dual band decoding that properly tries to match velocity and energy vectors in the low and high frequency bands. Or maybe the internal renderer should be scrapped altogether, it was merely created as a convenience and an external decoder[9] could be used in most if not all cases (for example Ambdec includes hand tuned configurations for 5.1 Ambisonics rendering[8]).
- The unfinished HRTF based back end should be finished and included in the distribution.
- It would also be interesting to explore the possibility of adding a Wave Field Synthesis back end. This would be difficult as it would imply a separate soundfile for each sound object, an approach that is at odds with the current "piece as a soundfile" Dlocsig / CLM system. *Path* objects, on the other hand, could easily generate the information to later do WFS rendering of the soundfiles.
- The Bezier curve fitting and rendering system for the *path* objects should be reconsidered to see if using a different type of curve fitting algorithm might produce better results. Bezier segment fitting can sometimes result in pathological behavior with some paths, specially with loops being created automatically. A more generic approach could use NURBs (Non-uniform rational B-splines) but fitting algorithms would have to be found.
- The tessellation algorithm described in Pulkki's VBAP paper[4] should be implemented so that grouping of speakers is automatically done.

ACKNOWLEDGMENT

Thanks to Bill Schottstaedt for creating, maintaining and expanding the wonderful CLM environment.

REFERENCES

[1] Common Lisp Music (CLM): http://ccrma.stanford.edu/software/clm/

[2] Bill Schottstaedt, "CLM: Music V Meets Common Lisp," *Computer Music Journal* 18(2):30-37, 1994.

[3] John. Chowning, "The simulation of moving sound sources," *Journal of the Audio Engineering Society*, vol. 19, no. 1, pp. 26, 1971.

[4] V. Pulkki, "Virtual sound source positioning using vector base amplitude panning", *Journal of the Audio Engineering Society*, 45(6) pp. 456-466, June 1997.

[5] Michael A. Gerzon, "Periphony: With-Height Sound Reproduction", *Journal of the Audio Engineering Society*, 1973, 21(1):210

[6] Dave Malham, "http://www.york.ac.uk/inst/mustech/3d_audio/higher_order_ambisonics.pdf", 2003

[7] Jerome Daniel, "Reprsentation de champs acoustiques, application la transmission et la reproduction de scnes sonores complexes dans un contexte multimdia", Thse de doctorat de lUniversit Paris 6, 2001

[8] Bruce Wiggins, "An Investigation into the Real-time Manipulation and Control of Three-dimensional Sound Fields", University of Derby Doctoral Thesis, 2004

[9] Fons Adriansen, Ambdec, an open source Ambisonics decoder, "http://www.kokkinizita.net/linuxaudio/downloads/ambdec-manual.pdf"

[10] Fernando Lopez-Lezcano, "A Four Channel Dynamic Sound Location System", *The Japan Music and Computer Science Society (JMACS) 1992 Summer Symposium*, 1992

[11] Fernando Lopez-Lezcano, "A dynamic spatial sound movement kit", *International Computer Music Conference (ICMC)*, 1994

[12] Julius Smith, "http://www-ccrma.stanford.edu/ jos/kna/Experiences_Samson_Box.html"