Implementation of a Shape Synthesizer in Pure Data and Graphics Environment for Multimedia (GEM)

By James Tittle, II
tigital@mac.com

Introduction

Most 3D shapes are built using modelling programs that assemble complex shapes from primitive shapes, or by plotting some mathematical function. These methods are good for certain tasks, but don't necessarily lend themselves to freeform experimentation and discovery of new and interesting shapes.

Andrew Glassner first outlined a shape synthesizer in 1977. His shape synthesizer was conceived as a 3D graphic analog to the notion of a modular sound synthesizer. Due to the modularity of the "patcher" design and the need for message passing, Pure Data seems to be an ideal candidate for realizing these shape synthesizer ideas. This paper is an attempt at implementing many of his ideas in pure data, using GEM, Zexy, and other external libraries.

As a bit of background, we need to understand how an early modular sound synthesizer works in order to translate the function to the shape synthesizer. In a modular sound synthesizer, sounds are created by combining sound sources (such as oscillators and samples) with wave filters, delays, and other manipulations via a single generic electricity carrying wire. The complex interplay of the simple, individual modules can easily lead to novel sound forms not predictable from the behavior of any one module. This idea of simple manipulations leading to an unexpected final end is one of the main abstractions used in the shape synthesizer. It is also important to note the idea of the wire/patch cord, carrying a single form of data, which is used to connect different modules.

So in essence, the shape synthesizer is a collection of modules that speak a common language: all of the modules communicate by accepting and/or outputting vectors, which can be considered as a 3D point (x, y, z). A number of these vectors filter thru the assembled modules, warping along the way to the output, where they are drawn onto the screen. So let's look at the modules.

Module Descriptions

Following are individual summaries of the basic abstractions that comprise the shape synthesizer. If an input does not have anything connected to it, a value of 0 is used.

Clock

The clock module is the beginning of all the vectors: for every evaluation of the shape synthesizer network, it produces a unique vector. Conceptually, this determines the "parameter space" for the shape: each "tick" of the clock outputs one "slice" or section of the shape. There are two variations of

Clock: Linear Clock and Boxed Clock.

[clock_linear]

The [clock_linear] operates by outputting a vector that starts at (0, 0, 0), ends at (1, 1, 1), and advances by n steps along the line. For example, with *n*=10, each step would be 0.2, so the first vector output is (0, 0, 0), then (0.1, 0.1, 0.1), (0.2, 0.2, 0.2),...until (1, 1, 1), at which point the clock is reset to (0, 0, 0).

[clock_box]

The [clock_box] is useful for producing shapes which aren't functions of a single parameter, liketo be made out of sub-shapes. Instead of advancing along a line, a triply-nested loop is performed using a third tuple (nx, ny, nz), where nx steps are taken in the X axis, followed by ny steps along the Y axis, and finally nz steps along the Z axis. For example, if our third tuple is (10, 10, 1), the clock advances ten steps through X before incrementing Y, and this is repeated 10 steps in Y before incrementing the Z value.

Vector Processors

Vector-Controlled Translator: [vecTranslator]

The vector-controlled translator takes two input vectors and sums them by component.
(x, y, z) = (x0+x1, y0+y1, z0+z1)

Vector-Controlled Scaler: [vecScaler]

The vector-controlled scaler takes two input vectors and multiplies them component by component.
(x, y, z) = (x0x1, y0y1, z0z1)

Vector-Controlled Rotator: [vecRotator]

The vector-controlled rotator takes three inputs: The first input on the left is considered a signal, or 3D point; the middle input is considered the base, and the right-most input is considered the axis. Taken together, the axis determines the direction and length of a line that passes through the base point. The signal is then rotated around the axis/base defined line by an angle equal to the length of the axis (interpreted in radians). Thus an axis of length 2*pi will rotate the signal point one revolution around the axis.

Vector-Controlled Cross: [vecCross]
The vector-controlled cross takes two input vectors and performs a right-handed cross-product, which produces an output vector that is perpendicular to the plane in which the first two lie:
(x, y, z) = (y0z1 - z0y1, z0x1 - x0z1, x0y1 - y0x1)

[route]
A [route] abstraction takes one input vector and allows a remapping of it's

values to the output. For example, (x, y, z) could be remapped to (x, z, y) or (z, x, y).

[genop]

The [genop] abstraction is a general operator. It accepts one input (although it could be expanded as needed) and unpacks the list, thereby allowing for arbitrary computations to be performed on individual elements.

Input/Output/Miscellaneous Modules

Input: [$1 $2 $3<, [$1 0 0<, [0 $1 0<, etc.

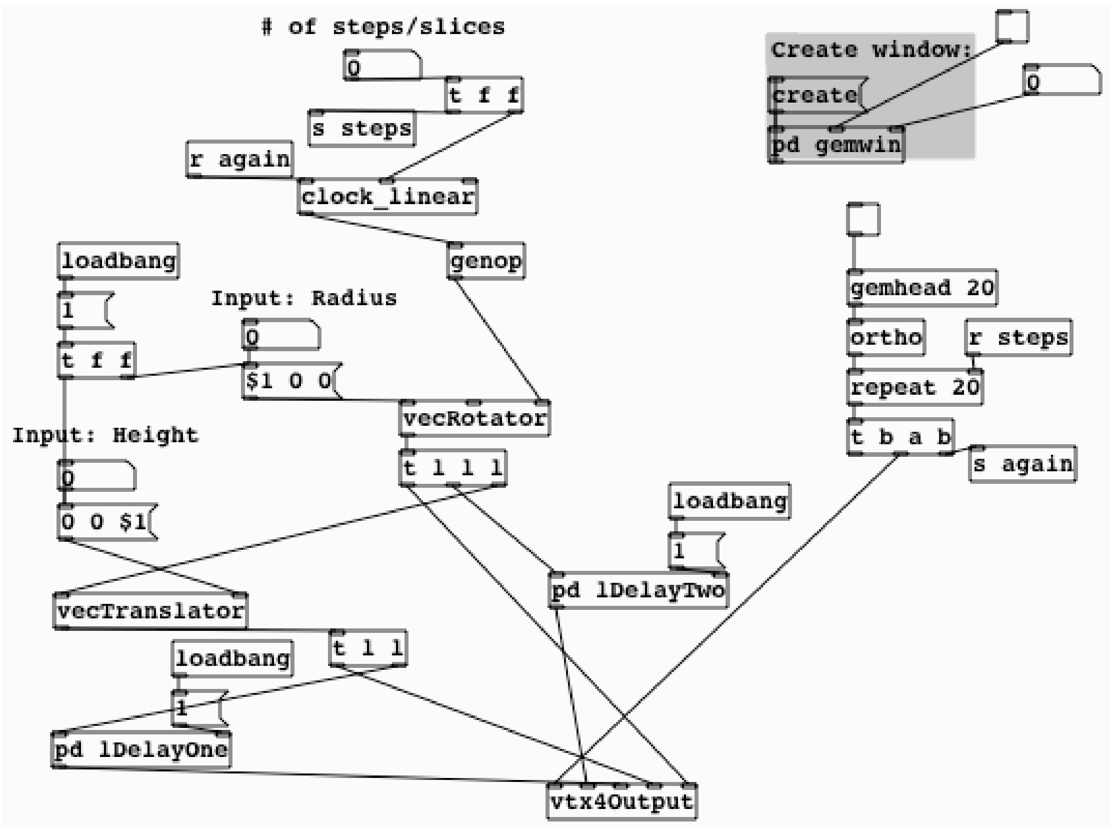
The list message is used here to allow the generation of vector members: this allows for the flexibility of interactive input, reading inputs from a file, or mathematical expressions for each component.

VertexOutput: [vtx3Output], [vtx4Output]

This abstraction is variable depending on the desired format of the drawing. It is comprised of the OpenGL wrappers for glVertex*() and anyother desired drawing calls. The left inlet will accept a GemState bang, while there could be a variable number of right inlets. Each of these inlets accepts one vector which is interpreted as a vertex; the abstraction can be setup to accept three vertices and draw triangles, or four vertices and draw quads. It is important to get the order of the vertices correct (some experimentation may be required), or unpredictable drawing will occur!

Delay: [vecDelay]

The [vecDelay] abstraction accepts two inputs: the left inlet is a vector/ list, and the right inlet is a number representing the length of the delay. This abstraction holds a number of vectors equal to it's "delay length", and outputs only when "delay length"+1 vectors have accumulated. For example, if the "delay length" is 1, then the first vector to arrive will be stored and produce no output. When a second vector arrives it is stored and the first vector is output.



Example: a cylinder

The cylinder patch is a simple example of how the shape synthesizer abstractions can be used. It starts with [clock_linear] being fed into [genop], which converts the clock impulse to (0, 0, 2pi*z). This is then fed into the right (or axis) input of [vecRotator] (whose middle/base input is zero), along with an [input] module that outputs (x, 0, 0) to the left/point input, which represents the radius of the cylinder. [vecRotator] then outputs three copies of a new vector: one goes to [vecTrans], one goes to a 1 step [vecDelay], and the third goes to the [vtx4Output]. The [vecTrans] also accepts an input vector that establishes the height of the cylinder (0, 0, z), and outputs two copies of it's new multiplied vector: one to a 1 step [vecDelay] and the other to [vtx4Output]. So for the first clock tick we only get two vertices produced: in practice this is not a noticeable glitch (it only occurs in the first frame generated), but could be eliminated by introducing a "pre-roll" that primes the delays with some value. After the second clock tick we now get four vertices delivered to [vtx4Output]. The clock is hooked into the [gemhead] system by a [repeat] module (set to the number of steps in the clock). This has the effect of producing one model for each bang (or frame) of the [gemhead].

Conclusions and Future Directions

As seen above, Pure Data and GEM are more than adequate to implement the basics of a shape synthesizer. However, these are only the basics, and like sound synthesis, there are many other methods for manipulation of the data. One can easily imagine other matrix transforms (such as dot products) implemented as modules. The clock module is ripe for change, which could lead to fundamentally new shapes, due to it's crucial role in establishing the initial vector space.

The history of sound synthesis could be looked to for new methods to implement. For instance, in sound synthesis there is the idea of frequency modulation, where the frequency of one sound is used as the input control. This is readily translatable to the shape synthesizer because we can plug the output of any module to any other input and thereby get shape modulation.

In the present iteration, the shape synthesizer outputs via individual calls to vertex drawing functions: this immediate mode of drawing does not scale well in OpenGL. Recently, much progress has been made for a more efficient method of drawing using vertex arrays. It is conceivable to imagine a [vertex_sink] module which would collect an entire shape's worth of vertices and then quickly pass that to the GPU for further processing. Along with this, the GPU may be further utilized by implementing many of the individual modules in a series of vertex programs: adding support for ARB_vertex_program and ARB_fragment_program and shading languages such as GLSL and Cg are currently the focus of much GEM development. The [vtx4Outputs] could also be expanded to accept colors, texture coordinates, or generate other shading/normal information.

Finally, it would be desirable to include a method of saving generated shapes to some 3D file format such as OBJ or 3DS.

Special Thanks to IOhannes Zmoelnig, Chris Clepper and Frank Barknect for valuable input and sanity checking.

References

Barknecht, Frank. listDelay.pd abstraction, personal communication, August 31, 2004.

Glassner, Andrew, ed., Graphics Gems I. 1990. Boston: Academic Press.

Glassner, Andrew. "A Shape Synthesizer", IEEE Computer Graphics & Applications, 17(3), pgs. 40-51, May/June 1997.

Hill, F. S. Computer Graphics Using Open GL, 2nd Ed. 2001. NJ: Prentice Hall, 2001.

Maillot, Patrick-Gilles. "Using Quaternions for Coding 3D Transformations," in GEMS I, pp. 498-515.