Fluxus documentation

Jaromil's Journal of Musings

November 30, 2008

Contents

Contents **1**
0.1 Fluxus Documentation 0.14 . . . 1
0.2 Introduction . . . 1
0.3 UserGuide . . . 2
0.4 Scheme . . . 3
0.5 State Machine . . . 7
0.6 About Primitives . . . 9
0.7 Polygon Primitives . . . 11
0.8 Nurbs Primitives . . . 14
0.9 Exotic Primitives . . . 14
0.10 Deforming . . . 17
0.11 Loading And Saving . . . 20
0.12 Shaders . . . 20
0.13 Turtle Builder . . . 22
0.14 Making Movies . . . 22
0.15 Fluxus In DrScheme . . . 23
0.16 Fluxus Scratchpad And Modules . . . 24

0.1 Fluxus Documentation 0.14

by Dave Griffiths

This is some detailed text documentation about fluxus and the general concepts it uses. It's quite dense, so I recommend trying the examples and using this as reference as you learn scheme and work your way through.

The text is taken from the pawfal wiki[1] look there for the current version (which may not match this version of fluxus).

0.2 Introduction

A graphical livecoding environment for Scheme. Builds for Linux or OSX, and released under the GPL licence.

[1] http://www.pawfal.org/index.php?page=FluxusDocumentation

Fluxus reads live audio or OSC network messages which can be used as a source of animation data for realtime performances or installations. Keyboard or mouse input can also be read for simple games development, and a physics engine is included for realtime simulations of rigid body dynamics.

The built in scheme code editor runs on top of the renderer, which means you can edit the scripts while they are running. As well as making livecoding possible, it's also gives you a very fast feedback way of experimenting or learning about graphics and animation.

Fluxus lends itself to procedural modelling and animation. There is an experimental procedural modelling tool, and full support for texturing, basic material properties and hardware shading.

Fluxus consists of an application, which embeds a Scheme interpreter, called the fluxus scratchpad, and a set of Scheme modules which extend an interpreter with graphics commands.

Quickstart

When you start fluxus, you will see the welcome text and a prompt - this is called the repl, or console. Generally fluxus scripts are written in text buffers, which can be switched to with ctrl and the numbers keys. Switch to the first one of these by pressing ctrl-1 (ctrl-0 switched you back to the fluxus console).

Now try entering the following command.

```
(build-cube)
```

Now press **F5 (or ctrl-e)** - the script will be executed, and a white cube should appear in the centre of the screen. Use the mouse and to move around the cube, pressing the buttons to get different movement controls.

To animate this cube, it will have to be called a little differently:

```
; buffersize and samplerate need to match jack's
    (start-audio "jack-port-to-read-sound-from" 256 44100)

(define (render)
    (colour (vector (gh 1) (gh 2) (gh 3)))
            (draw-cube))

(every-frame (render))
```

Briefly, the (every-frame) function takes a function which is called once per frame by fluxus's internal engine. In this case it calls a function that sets the current colour using harmonics from the incoming sound with the (gh) - get harmonic - function; and draws a cube. Note that this time we use (draw-cube) not (build-cube). The difference will be explained below.

If everything goes as planned, and the audio is connected with some input - the cube will flash in a colourful manner along with the sound.

Now go and have a play with the examples. Load them by pressing ctrl-l or on the commandline, by entering the examples directory and typing fluxus followed by the script filename.

0.3 UserGuide

When using the fluxus scratchpad, the idea is that you only need the one window to build scripts, or play live. f5 (or ctrl-e) is the key that runs the script when you are ready. Selecting some text (using shift) and pressing f5 will execute the selected text only. This is handy for reevaluating functions without running the whole script each time.

Camera control: the camera is controlled by moving the mouse and pressing mouse buttons: (left mouse button = rotate), (middle mouse button = move), (right mouse button = zoom).

Workspaces

The script editor allows you to edit 9 scripts simultaneously by using workspaces. To switch workspaces, use ctrl+number key. Only one can be run at once though, hitting f5 will execute the currently active workspace script. Scripts in different workspaces can be saved to different files, press ctrl-s to save or ctrl-d to save-as and enter a new filename (the default filename is temp.scm).

The REPL

If you press ctrl and 0, instead of getting another script workspace, you will be presented with a Read Evaluate Print Loop interpreter, or repl for short. This is really just an interactive interpreter similar to the commandline, where you can enter scheme code for immediate evaluation. This code is evaluated in the same interpreter as the other scripts, so you can use the repl to debug or inspect global variables and functions they define. This window is also where error reporting is printed, along with the terminal window you started fluxus from.

Keyboard commands

```
- ctrl-f : Fullscreen mode.
- ctrl-w : Windowed mode.
- ctrl-h : Hide/show the editor.
- ctrl-l : Load a new script (navigate with cursors and return).
- ctrl-s : Save current script.
- ctrl-d : Save as - current script (opens a filename dialog).
- ctrl-1 to 9 : Switch to selected workspace.
- ctrl-0 : Switch to the REPL.
- F3 : Resets the camera.
- F5/ctrl-e : Execute the selected text, or all if none is selected.
              ctrl-e might be easier for mac users,
              as volume is usually bound to the f5 key.
- F9 : Randomise the text colour (aka the panic button)
- Escape : Editor panic button, resets the cursor if it gets stuck
               (let me know if you need to use this)
```

0.4 Scheme

Scheme is a programming language invented by Jerald J. Sussman and Guy L. Steel Jr. in 1975. Scheme is based on another language - Lisp, which dates back to the fifties. It is a high level language, which means it is biased towards human, rather than machine understanding. The fluxus scratchpad embeds a Scheme interpreter (it can run Scheme programs) and the fluxus modules extend the Scheme language with commands for 3D computer graphics.

This chapter gives a very basic introduction to Scheme programming, and a fast path to working with fluxus - enough to get you started without prior programming experience, but I don't explain the details very well. For general scheme learning, I heartily recommend the following books (two of which have the complete text online):

The Little Schemer Daniel P. Friedman and Matthias Felleisen

How to Design Programs An Introduction to Computing and Programming Matthias Felleisen Robert Bruce Findler Matthew Flatt Shriram Krishnamurthi Online: `http://www.htdp.org/2003-09-26/Book/`

Structure and Interpretation of Computer Programs Harold Abelson and Gerald Jay Sussman with Julie Sussman Online: `http://mitpress.mit.edu/sicp/full-text/book/book.html`

We'll start by going through some language basics, which are easiest done in the fluxus scratchpad using the console mode - launch fluxus and press ctrl 0 to switch to console mode.

Scheme as calculator

Languages like Scheme are composed of two things - operators (things which do things) and values which operators operate upon. Operators are always specified first in Scheme, so to add 1 and 2, we do the following:

```
fluxus> (+ 1 2)
    3
```

This looks pretty odd to begin with, and takes some getting used to, but it means the language has less rules and makes things easier later on. It also has some other benefits, in that to add 3 numbers we can simply do:

```
fluxus> (+ 1 2 3)
    6
```

It is common to "nest" the brackets inside one another, for example:

```
fluxus> (+ 1 (* 2 3))
    7
```

Naming values

If we want to specify values and give them names we can use the Scheme command "define":

```
fluxus> (define size 2)
fluxus> size
    2
fluxus> (* size 2)
    4
```

Naming is arguably the most important part of programming, and is the simplest form of what is termed "abstraction" - which means to separate the details (e.g. the value 2) from the meaning - size. This is not important as far as the machine is concerned, but it makes all the difference to you and other people reading code you have written. In this example, we only have to specify the value of size once, after that all uses of it in the code can refer to it by name - making the code much easier to understand and maintain.

Naming procedures

Naming values is very useful, but we can also name operations (or collections of them) to make the code simpler for us:

```
fluxus> (define (square x) (* x x))
fluxus> (square 10)
    100
fluxus> (square 2)
    4
```

Look at this definition carefully, there are several things to take into account. Firstly we can describe the procedure definition in English as: To (define (square of x) (multiply x by itself)) The "x" is called an argument to the procedure, and like the size define above - it's name doesn't matter to the machine, so:

```
fluxus> (define (square apple) (* apple apple))
```

Will perform exactly the same work. Again, it is important to name these arguments so they actually make some sort of sense, otherwise you end up very confused. Now we are abstracting operations (or behaviour), rather than values, and this can be seen as adding to the vocabulary of the Scheme language with our own words, so we now have a square procedure, we can use it to make other procedures:

```
fluxus> (define (sum-of-squares x y)
                (+ (square x) (square y)))
fluxus> (sum-of-squares 10 2)
    104
```

The newline and whitespace tab after the define above is just a text formatting convention, and means that you can visually separate the description and it's argument from the internals (or body) of the procedure. Scheme doesn't care about whitespace in it's code, again it's all about making it readable to us.

Making some shapes

Now we know enough to make some shapes with fluxus. To start with, leave the console by pressing ctrl-1 - you can go back at any time by pressing ctrl-0. Fluxus is now in script editing mode. You can write a script, execute it by pressing F5, edit it further, press F5 again... this is the normal way fluxus is used.

Enter this script:

```
(define (render)
    (draw-cube))

(every-frame (render))
```

Then press F5, you should see a cube on the screen, drag the mouse around the fluxus window, and you should be able to move the camera - left mouse for rotate, middle for zoom, right for translate.

This script defines a procedure that draws a cube, and calls it every frame - resulting in a static cube.

You can change the colour of the cube like so:

```
(define (render)
    (colour (vector 0 0.5 1))
        (draw-cube))

(every-frame (render))
```

The colour command sets the current colour, and takes a single input - a vector. Vectors are used a lot in fluxus to represent positions and directions in 3D space, and colours - which are treated as triplets of red green and blue. So in this case, the cube should turn a light blue colour.

Transforms

Add a scale command to your script:

```
(define (render)
    (scale (vector 0.5 0.5 0.5))
        (colour (vector 0 0.5 1))
        (draw-cube))

(every-frame (render))
```

Now your cube should get smaller. This might be difficult to tell, as you don't have anything to compare it with, so we can add another cube like so:

```
(define (render)
    (colour (vector 1 0 0))
        (draw-cube)
        (translate (vector 2 0 0))
        (scale (vector 0.5 0.5 0.5))
        (colour (vector 0 0.5 1))
        (draw-cube))

(every-frame (render))
```

Now you should see two cubes, a red one, then the blue one, moved to one side (by the translate procedure) and scaled to half the size of the red one.

```
(define (render)
    (colour (vector 1 0 0))
        (draw-cube)
        (translate (vector 2 0 0))
        (scale (vector 0.5 0.5 0.5))
        (rotate (vector 0 45 0))
        (colour (vector 0 0.5 1))
    (draw-cube))

(every-frame (render))
```

For completeness, I added a rotate procedure, to twist the blue cube 45 degrees.

Recursion

To do more interesting things, we will write a procedure to draw a row of cubes. This is done by recursion, where a procedure can call itself, and keep a record of how many times it's called itself, and end after so many iterations.

In order to stop calling our self as a procedure, we need to take a decision - we use cond for decisions.

```
(define (draw-row count)
    (cond
        ((not (zero? count))
            (draw-cube)
                (translate (vector 1.1 0 0))
                (draw-row (- count 1)))))

(every-frame (draw-row 10))
```

Be careful with the brackets - the fluxus editor should help you by highlighting the region each bracket corresponds to. Run this script and you should see a row of 10 cubes. You can build a lot out of the concepts in this script, so take some time over this bit.

cond is used to ask questions, and it can ask as many as you like - it checks them in order and does the first one which is true. In the script above, we are only asking one question, (not (zero? count)) - if this is true, if count is anything other than zero, we will draw a cube, move a bit and then call our self again. Importantly, the next time we call draw-row, we do so with one taken off count. If count is 0, we don't do anything at all - the procedure exits without doing anything.

So to put it together, **draw-row** is called with count as 10 by **every-frame**. We enter the draw-row function, and ask a question - is count 0? No - so carry on, draw a cube, move a bit, call draw-row again with count as 9. Enter draw-row again, is count 0? No, and so on. After a while we call draw-row with count as 0, nothing happens - and all the other functions exit. We have drawn 10 cubes.

Recursion is a very powerful idea, and it's very well suited to visuals and concepts like self similarity. It is also nice for quickly making very complex graphics with scripts not very much bigger than this one.

Animation

Well, now you've got through that part, we can quite quickly take this script and make it move.

```
(define (draw-row count)
    (cond
        ((not (zero? count))
            (draw-cube)
                (rotate (vector 0 0 (* 45 (sin (time)))))
                (translate (vector 1.1 0 0))
```

```
                (draw-row (- count 1)))))

(every-frame (draw-row 10))
```

time is a procedure which returns the time in seconds since fluxus started running. sin converts this into a sinewave, and the multiplication is used to scale it up to rotate in the range of -45 to +45 degrees (as sin only returns values between -1 and +1). Your row of cubes should be bending up and down. Try changing the number of cubes from 10, and the range of movement by changing the 45.

More recursion

To give you something more visually interesting, this script calls itself twice - which results in an animating tree shape.

```
(define (draw-row count)
    (cond
        ((not (zero? count))
            (translate (vector 2 0 0))
                (draw-cube)
                (rotate (vector (* 10 (sin (time))) 0 0))
                (with-state
                        (rotate (vector 0 25 0))
                        (draw-row (- count 1)))
            (with-state
                        (rotate (vector 0 -25 0))
                        (draw-row (- count 1))))))

(every-frame (draw-row 10))
```

For an explanation of with-state, see the next section.

0.5 State Machine

The state machine is the key to understanding how fluxus works, all it really means is that you can call functions which change the current context which has an effect on subsequent functions. This is a very efficient way of describing things, and is built on top of the OpenGl api, which works in a similar way. For example, in a function called every frame:

```
(colour (vector 1 0 0))
    (draw-cube)
    (colour (vector 0 1 0))
    (draw-cube)
```

Will draw a red cube, then a green cube (in this case, you can think of the (colour) call as changing a pen colour before drawing something). States can also be stacked, for example:

```
(colour (vector 1 0 0))
    (with-state
     (colour (vector 0 1 0))
     (draw-cube))
(draw-cube)
```

Will draw a green, then a red cube. The (with-state) isolates a state and gives it a lifetime, indicated by the brackets (so changes to the state inside are applied to the build-cube inside, but do not affect the build-cube afterwards). Its useful to use the indentation to help you see what is happening.

The Scenegraph

Both examples so far have used what is known as immediate mode, you have one state stack, the top of which is the current context, and everything is drawn once per frame. fluxus contains a structure known as a scenegraph for storing objects and their render states.

Time for another example:

```
(colour (vector 1 0 0))
    (build-cube)
    (colour (vector 0 1 0))
    (build-cube)
```

The only difference between this and the first example is the use of (build-cube) instead of (draw-cube). the build functions create a primitive object, copy the current renderstate and add the information into the scenegraph in a container called a scenenode.

The **build-** functions return object ID's (just numbers really) which enable you to do things to the scene node after it's been created. you can now specify objects like this:

```
(define myob (build-cube))
```

The cube will now be persistent in the scene until destroyed with

```
(destroy myob)
```

with-state returns the result of it's last expression, so to make new primitives with state you set up, you can do this:

```
(define my-object (with-state
    (colour (vector 1 0 0))
        (scale (vector 0.5 0.5 0.5))
        (build-cube)))
```

If you want to modify a objects renderstate after it's been loaded into the scenegraph, you can use with-primitive to set the current context to that of the object. This allows you to animate objects you have built, for instance:

```
% build some cubes

(colour (vector 1 1 1))
    (define obj1 (build-cube))
    (define obj2 (with-state
           (translate (vector 2 0 0))
           (build-cube))

...

% in a function called per frame

(with-primitive obj1
           (rotate (vector 0 1 0)))

(with-primitive obj2
           (rotate (vector 0 0 1)))
```

The scenegraph also enables you to parent objects to one another, using the renderstate's parent setting. this is only effective before an object is loaded into the scenegraph, setting it afterwards via with-primitive will be ignored:

```
(colour (vector 1 1 1))
    (define a (build-cube))

(define b (with-state
           (parent a)
           (translate (vector 0 2 0))
           (build-cube)))

(define c (with-state
           (parent b)
           (translate (vector 0 2 0))
           (build-cube)))
```

Creates three cubes, all attached to each other in a chain. Transforms for object a will be passed down to b and c, transforms on b will effect c.

Destroying a object in such a hierarchy will in turn destroy all child objects parented to it.

> **Note on grabbing and pushing**: Fluxus also contains less well mannered commands for achieving the same results as with-primitive and with-state. These were used prior to version 0.14, so you may see mention of them in the documentation, or older scripts: (push)...(pop) is the same as (with-state), (grab myprim)...(ungrab) is the same as (with-primitive myprim ...)

0.6 About Primitives

Primitives are objects that you can render. There isn't really much else in a fluxus scene, except lights, a camera and lots of primitives.

Primitive state

The normal way to create a primitive is to set up some state which the primitive will use, then call it's build function and keep it's returned ID (using with-primitive) to modify it's state later on.

```
(define myobj (with-state
           (colour (vector 0 1 0))
           (build-cube))) ; makes a green cube

(with-primitive myobj
           (colour (vector 1 0 0))) ; changes it's colour to red
```

So primitives contain a state which describes things like colour, texture and transform information. This state operates on the primitive as a whole - one colour for the whole thing, one texture, shader pair and one transform. To get a little deeper and do more we need to introduce primitive data.

Primitive Data Arrays [aka. pdata]

A pdata array is a fixed size array of information contained within a primitive. Each pdata array has a name, so you can refer to it, and a primitive may contain lots of different pdata arrays (which are all the same size). Pdata arrays are typed - and can contain floats, vectors, colours or matrices. You can make your own pdata arrays, with names that you choose, or copy them in one command.

Some pdata is created when you call the build function. This automatically generated pdata is given single character names. Sometimes this automatically created pdata results in a primitive you can use straight away (in commands such as build-cube) but some primitives are only useful if pdata is setup and controlled by you.

In polygons, there is one pdata element per vertex - and a separate array for vertex positions, normals, colours and texture coordinates.

So, for example (build-sphere) creates a polygonal object with a spherical distribution of vertex point data, surface normals at every vertex and texture coordinates, so you can wrap a texture around the primitive. This data (primitive data, or pdata for short) can be read and written to inside a with-primitive corresponding to the current object.

```
(pdata-set! name vertnumber vector)
```

Sets the data on the current object to the input vector

```
(pdata-ref name vertnumber)
```

Returns the vector from the pdata on the current object

```
(pdata-size)
```

Returns the size of the pdata on the current object (the number of verts)

The name describes the data we want to access, for instance "p" contains the vertex positions:

```
(pdata-set! "p" 0 (vector 0 0 0))
```

Sets the first point in the primitive to the origin (not all that useful)

```
(pdata-set! "p" 0 (vadd (pdata-ref "p" 0) (vector 1 0 0)))
```

The same, but sets it to the original position + 1 in the x offsetting the position is more useful as it constitutes a deformation of the original point. (See Deforming, for more info on deformations)

Mapping, Folding

The pdata-set! and pdata-ref procedures are useful, but there is a more powerful way of deforming primitives. Map and fold relate to the scheme functions for list processing, it's probably a good idea to play with them to get a good understanding of what these are doing.

```
(pdata-map! procedure read/write-pdata-name read-pdata-name ...)
```

Maps over pdata arrays - think of it as a for-every pdata element, and writes the result of procedure into the first pdata name array.

An example, using pdata-map to invert normals on a primitive:

```
(define p (build-sphere 10 10))

(with-primitive p
    (pdata-map!
                (lambda (n)
                    (vmul n -1))
                "n"))
```

This is more concise and less error prone than using the previous functions and setting up the loop yourself.

```
(pdata-index-map! procedure read/write-pdata-name read-pdata-name ...)
```

Same as pdata-map! but also supplies the current pdata index number to the procedure as the first argument.

```
(pdata-fold procedure start-value read-pdata-name read-pdata-name ...)
```

This example calculates the centre of the primitive, by averaging all it's vertex positions together:

```
(let ((centre
    (with-primitive my-torus
                (vdiv (pdata-fold vadd (vector 0 0 0) "p") (pdata-size)))))
...)

(pdata-index-fold procedure start-value read-pdata-name read-pdata-name ...)
```

Same as pdata-fold but also supplies the current pdata index number to the procedure as the first argument.

Instancing

Sometimes retained mode primitives can be unwieldy to deal with. For instance, if you are rendering thousands of identical objects, or doing things with recursive graphics, where you are calling the same primitive in lots of different states - keeping track of all the IDs would be annoying to say the least.

This is where instancing is helpful, all you call is:

```
(draw-instance myobj)
```

Will redraw any given object in the current state (immediate mode). An example:

```
(define myobj (build-nurbs-sphere 8 10)) ; make a sphere

(define (render-spheres n)
           (cond ((not (zero? n))
           (with-state
                   (translate (vector n 0 0)) ; move in x
                           (draw-instance myobj))     ; stamp down a copy
           (render-spheres (- n 1)))))    ; recurse!

(every-frame (render-spheres 10)) ; draw 10 copies

** Built In Immediate Mode Primitives

To make life even easier than having to instance primitives, there are
some built  in primitives  that can be  rendered at any  time, without
being built:
<src>
     (draw-cube)
         (draw-sphere)
         (draw-plane)
         (draw-cylinder)
```

For example:

```
    (define (render-spheres n)
               (cond ((not (zero? n))
                       (with-state
                               (translate (vector n 0 0)) ; move in x
                               (draw-sphere))             ; render a new sphere
               (render-spheres (- n 1)))))    ; recurse!

     (every-frame (render-spheres 10)) ; draw 10 copies
```

These built in primitives are very restricted in that you can't edit them or change their resolution settings etc, but they are handy to use for quick scripts with simple shapes.

0.7 Polygon Primitives

Poly prims are the most versatile primitives in fluxus, and although they are all the same underlying type of primitive, there are many procedures used to build them: build-cube, build-sphere, build-torus, build-plane, build-seg-plane, build-cylinder, build-torus, build-polygons

The last one is useful if you are building your own shapes, the others are useful for giving you some preset shapes. All build-* functions return an id number which you can store and use to modify the primitive later on.

Pdata Types: - Positions: “p” vector - Normals: “n” vector - Texture coords: “t” vector - Vertex colours: “c” colour

Polygon topology and pdata

In the case of polygonal objects, the topology of the polygons dictates what faces are describe by the pdata vertex elements. A polygon primitive built from triangle lists will have the following ordering of it’s pdata:

Triangle list:

```
      0   1--2
    / |   |b/
   /  |   |/
  / a |   0
 2----1

pdata index 0 : face a vert 0
pdata index 1 : face a vert 1
pdata index 2 : face a vert 2
pdata index 3 : face b vert 0
pdata index 4 : face b vert 1
pdata index 5 : face b vert 2
```

Quad lists:

```
0---1         1----2
| a |        /     |
3---2       /   b  |
          0-------3

pdata index 0 : face a vert 0
pdata index 1 : face a vert 1
pdata index 2 : face a vert 2
pdata index 3 : face a vert 3
pdata index 4 : face b vert 0
pdata index 5 : face b vert 1
pdata index 6 : face b vert 2
pdata index 7 : face b vert 3
```

Triangle strip:

```
1---3---5
|\ b|\ d|
| \ | \ |
|a \|c \|
0---2---4

pdata index 0 : face a vert 0
pdata index 1 : face a vert 1 & face b vert 0
```

```
pdata index 2 : face a vert 2 & face b vert 1 & face c vert 0
pdata index 3 : face b vert 2 & face c vert 1 & face d vert 0
pdata index 4 : face c vert 2 & face d vert 1 & face e vert 0
pdata index 5 : face d vert 2 & face e vert 1 & face f vert 0
```

Triangle fan:

```
  1--2
  |a/ \
  |/ b \
  0-----3
  |\ c /
  |d\ /
  5--4

pdata index 0 : vert 0 for all faces
pdata index 1 : face a vert 1
pdata index 2 : face a vert 2 & face b vert 1
pdata index 3 : face b vert 2 & face c vert 1
pdata index 4 : face c vert 2 & face d vert 1
pdata index 5 : face d vert 2 & face e vert 1
```

Polygon (the easy one):

```
    0----1
   /     |
  /      |
 5       2
  \     /
   4---3

pdata index 0 : vert 0
pdata index 1 : vert 1
pdata index 2 : vert 2
pdata index 3 : vert 3
pdata index 4 : vert 4
pdata index 5 : vert 5
```

This lookup is the same for all the pdata on a particular polygon primitive - vertex positions, normals, colours and texture coordinates.

Although this implicit topology means the primitive is optimised to be very quick to render, it costs some memory as points are duplicated. This is a standard trade off, the most optimal poly topology are triangle strips, as the duplication gets less, depending on how long your strips get.

Built in topologies

The topologies for the various polygon primitives are as follows:

- build-cube: quad list
- build-plane: quad list
- build-sphere: triangle list
- build-cylinder: triangle list
- build-torus: triangle list

Indexed polygons

The other way of improving efficiency is converting polygons to indexed mode. Indexing adds a level of indirection to the model, so vertices forming different faces can share the same vertex data. You can set the index of a polygon manually with:

```
(with-primitive myobj
    (poly-set-index list-of-indices))
```

Or automatically - which is recommended with:

```
(with-primitive myobj
    (poly-convert-to-indexed))
```

This procedure compresses the polygonal object by finding duplicated or very close vertices and "gluing" them together to form one vertex and multiple index references. Indexing has a number of advantages, one that large models will take less memory - but the big deal is that deformation or other per-vertex calulations will be much quicker.

Problems with automatic indexing

As all vertex information becomes shared for coincident vertices, automatically indexed polygons can't have different normals, colours or texture coordinates for vertices at the same position. This means that they will have to be smooth and continuous with respect to lighting and texturing. This could be fixed in time, with a more complicated conversion algorithm.

0.8 Nurbs Primitives

NURBS are parametric curved patch surfaces. They are handled in a similar way to polygon primitives, except that instead of vertices, pdata elements represent control vertices of the patch. Changing a control vertex causes the mesh to smoothly blend the change across it's surface.
- build-nurbs-sphere - build-nurbs-plane
Pdata Types:
- Positions: "p" vector - Normals: "n" vector - Texture coords: "t" vector

Nurbs topology

Nurbs pdata are much simpler to deal with, as the topology is just a patch grid for both spheres and subdivided planes. For example:

```
(build-nurbs-plane 5 5)
```

or

```
(build-nurbs-sphere 5 5)
```

Would give you a topology like this:

```
0---1---2---3---4
|   |   |   |   |
5---6---7---8---9
|   |   |   |   |
10--11--12--13--14
|   |   |   |   |
15--16--17--18--19
|   |   |   |   |
20--21--22--23--24
```

In the case of the sphere, points 0,5,10,15 and 20 would be coincident with 4,9,14,19 and 24. Also points 0,1,2,3 and 4 will be coincident (at the 'north pole') while the bottom row 20,21,22,23 and 24 would be coincident at the 'south pole'.

0.9 Exotic Primitives

'Exotic' primitives are just the leftovers after we have covered polygons and nurbs. They are generally a bit more specific, and most require some pdata editing to be useful - ie there are no preset forms which can be used with them, unlike polygons or nurbs.

- build-line
- build-text
- build-locator
- build-pixels
- build-blobby

Particle primitives

Particle primitives use the pdata elements to represent a point in screen space, or a camera facing sprite which can be textured - depending on the render hints. This primitive is useful for lots of different effects including, water, smoke, clouds and explosions.

```
(build-particles num-particles)
```

PData Types: - Positions: "p" vector - Colours: "c" colour - Sizes: "s" vector

Geometry hints

Particle primitives default to camera facing sprites, these are actually faster than points in tests on my machine - but they look quite different. To switch to points:

```
(with-primitive myparticles
    (hint-none)    ; turns off solid, which is default sprite mode
        (hint-points)) ; turn on points rendering
```

If you also enable (hint-anti-alias) you may get circular points, depending on your OpenGL setup - these can be scaled in pixel space using (point-width). You can scale the sprites in x and y using the "s" pdata array.

Line primitives

Line primitives are similar to particles in that they are either hardware rendered lines or camera facing textured quads. This primitive is useful for ribbon style effects, and draws a single line with each pdata element corresponding to a vertex point on the line. The texture is stretched along the line from the start to the end, and widthwise across the line. The width can be set per vertex to change the shape of the line.

```
(build-line num-points)
```

PData Types:

- Positions: "p" vector
- Colours: "c" colour
- Width: "w" float

Geometry hints

Line primitives default to the camera facing quads. To switch to points:

```
(with-primitive myline
    (hint-none) ; turns off solid, which is default mode
        (hint-wire)) ; turn on lines rendering
```

The wireframe lines can be scaled in pixel space using (line-width). The lines also pick up the colour pdata information.

Text primitive

Text primitives are not really well supported, but allow you to create text based on texture fonts. The font assumed to be non proportional - there is an example font shipped with fluxus.

```
(texture (load-texture "font.png"))
(build-text text-string)
```

The plan is to replace this with a true type primitive, let me know if you need this.

Locator primitive

The locator is a null primitive, as it doesn't render anything. Locators are useful for various tasks, you can use them as a invisible scene node to group primitive under. They are also used to build skeletons for skinning. To view them, you can turn on hint-origin, which draws an axis representing their transform.

```
(hint-origin)
(build-locator)
```

Pixel primitive

A pixel primitive is used for making procedural textures, which can then be applied to other primitives. For this reason, pixel primitives probably wont be rendered much, but you can render them to preview the texture on a flat plane.

```
(pixel-primitive width height)
```

PData Types:

- Colour: "c" vector
- Alpha: "a" float

Extra pixel primitive commands

A pixel primitive's pdata corresponds to pixel values in a texture, you write to them to make procedural texture data. The pixel primitive comes with some extra commands:

```
(pixels-upload pixelprimitiveid-number)
```

Uploads the texture data, you need to call this when you've finished writing to the pixelprim, and while it's grabbed.

```
(pixels->texture pixelprimitiveid-number)
```

Returns a texture you can use exactly like a normal loaded one.

See the examples for some procedural texturing. It's important to note that creating textures involves a large amount of processing time, so you don't want to plan on doing something per-pixel/per-frame for large textures. The pdata-func extensions could be used to help here in the future. Let me know if you want to know more about this.

Blobby Primitives

Blobby primitives are a higher level implicit surface representation in fluxus which is defined using influences in 3 dimesional space. These influences are then summed together, and a particular value is "meshed" (using the marching cubes algorithm) to form a smooth surface. The influences can be animated, and the smooth surface moves and deforms to adapt, giving the primitive it's blobby name. build-blobby returns a new blobby primitive. Numinfluences is the number of "blobs". Subdivisions allows you to control the resolution of the surface in each dimension, while boundingvec sets the bounding area of the primitive in local object space. The mesh will not be calculated outside of this area. Influence positions and colours need to be set using pdata-set.

```
(build-blobby numinfluences subdivisionsvec boundingvec)
```

See the examples for working blobby scripts.
PData Types:

- Position: "p" vector
- Strength: "s" float
- Colour: "c" vector

0.10 Deforming

Deformation in this chapter signifies various operations. It can involve changing the shape of a primitive in a way not possible via a transform (ie bending, warping etc) or modifying texture coordinates or colours to achieve a per-vertex effect. Deformation in this way is also the only way to get particle primitives to do anything interesting.

Deforming is all about pdata, so, to deform an entire object, you do something like the following:

```
(hint-unlit) (hint-wire) (line-width 4)

(define myobj (build-sphere 10 10))

(with-primitive myobj
    (pdata-map!
        (lambda (p)
                    ; add a small random vector to the original point
                    (vadd (vmul (vector (flxrnd) (flxrnd) (flxrnd)) 0.1)))
        "p"))
```

qWhen deforming geometry, moving the positions of the vertices is not usually enough, the normals will need to be updated for the lighting to work correctly.

```
(recalc-normals smooth)
```

Will regenerate the normals for polygon and nurbs primitives based on the vertex positions. not particularly fast (it is better to deform the normals in your script if you can). If smooth is 1, the face normals are averaged with the coincident face normals to give a smooth appearance.

When working on polygon primitives fluxus will cache certain results, so it will be a lot slower on the first calculation than subsequent calls on the same primitive.

User PData

As well as the standard information that exists in primitives, fluxus also allows you to add your own per vertex data to any primitive. User pdata can be written or read in the same way as the built in pdata types.

```
(pdata-add name type)
```

Where name is a string with the name you wish to call it, and type is a one character string consisting of:

- f : float data
- v : vector data
- c : colour data
- m : matrix data

```
(pdata-copy source destination)
```

This will copy a array of pdata, or overwrite an existing one with if it already exists.

So, adding your own storage for data on primitives means you can use it as a fast way of reading and writing data, even if the data doesn't directly affect the primitive.

An example of a particle explosion:

```
% setup the scene
(clear)
(show-fps 1)
(point-width 4)
(hint-anti-alias)

% build our particle primitive
(define particles (build-particles 1000))

% set up the particles
(with-primitive particles
    (pdata-add "vel" "v") ; add the velocity user pdata of type vector
    (pdata-map! ; init the velocities
        (lambda (vel)
            (vmul (vsub (vector (flxrnd) (flxrnd) (flxrnd))
                        (vector 0.5 0.5 0.5))
                  0.1))
        "vel")
    (pdata-map! ; init the colours
        (lambda (c)
            (vector (flxrnd) (flxrnd) 1 0))
        "c"))

(blur 0.1)

% a procedure to animate the particles
(define (animate)
    (with-primitive particles
        (pdata-map!
            (lambda (vel) (vadd vel (vector 0 -0.001 0)))
            "vel")
        (pdata-map! vadd "p" "vel")))

(every-frame (animate))
```

PData Operations

Pdata Operations are a optimisation which takes advantage of the nature of these storage arrays to allow you to process them with a single call to the scheme interpreter. This makes deforming primitive much faster as looping in the scheme interpreter is slow, and it also simplifies your scheme code.

```
(pdata-op operation pdata operand)
```

Where operation is a string identifier for the intended operation (listed below) and pdata is the name of the target pdata to operate on, and operand is either a single data (a scheme number or vector (length 3,4 or 16)) or a name of another pdata array.

If the (update) and (render) functions in the script above are changed to the following:

```
(define (update)
     (pdata-op "+" "vel" (vector 0 -0.002 0))
      ; add this vector to all the velocities
    (pdata-op "+" "p" "vel"))
      ; add all the velocities to all the positions

(define (render)
    (with-primitive ob
        (update)))
```

On my machine, this script runs over 6 times faster than the first version.

(pdata-op) can also return information to your script from certain functions called on entire pdata arrays.

Pdata operations:

- "+" : addition
- "*" : multiplication
- "sin" : writes the sine of one float pdata array into another
- "cos" : writes the cosine of one float pdata array into another
- "closest" : treats the vector pdata as positions, and if given a single vector, returns the closest position to it - or if given a float, uses it as a index into the pdata array, and returns the nearest position.

For most pdata operations, the vast majority of the combinations of input types (scheme number, the vectors or pdata types) will not be supported, you will receive a rather cryptic runtime warning message if this is the case.

PData functions

PData ops are useful, but I needed to expand the idea into something more complicated to support more interesting deformations like skinning. This area is messy, and somewhat experimental - so bear with me, it should solidify in future.

PData functions (pfuncs) range from general purpose to complex and specialised operations which you can run on primitives. All pfuncs share the same interface for controlling and setting them up. The idea is that you make a set of them at startup, then run them on one or many primitives later on per-frame.

```
(make-pfunc pfunc-name-symbol))
```

Makes a new pfunc. Takes the symbol of the names below, eg

```
(make-pfunc 'arithmetic)

(pfunc-set! pfuncid-number argument-list)
```

Sets arguments on a primitive function. The argument list consists of symbols and corresponding values.

(pfunc-run id-number) Runs a primitive function on the current primitive.

Look at the skinning example to see how this works.

Pfunc types

All pfunc types and arguments are as follows:

arithmetic

For applying general arithmetic to any pdata array

- operator string : one of add sub mul div
- src string : pdata array name
- other string : pdata array name (optional)
- constant float : constant value (optional)
- dst string : pdata array name

genskinweights

Generates skinweights - adds float pdata called "s1" -> "sn" where n is the number of nodes in the skeleton - 1

- skeleton-root primid-number : the root of the bindpose skeleton for skinning - sharpness float : a control of how sharp the creasing will be when skinned

skinweights->vertcols A utility for visualising skinweights for debugging. no arguments

skinning

Skins a primitive - deforms it to follow a skeleton's movements. Primitives we want to run this on have to contain extra pdata - copies of the starting vert positions called "pref" and the same for normals, if normals are being skinned, called "nref".

- skeleton-root primid-number : the root primitive of the animating skeleton - bindpose-root primid-number : the root primitive of the bindpose skeleton - skin-normals number : whether to skin the normals as well as the positions

Using PData to build your own primitives

The function (build_polygons) allows you to build empty primitives which you can use to either build more types of procedural shapes than fluxus supports naively, or for loading model data from disk. Once these primitives have been constructed they can be treated in exactly the same way as any other primitive, ie pdata can be added or modified, and you can use (recalc-normals) etc.

0.11 Loading And Saving

Fluxus has some support for loading and saving 3D data. This area is currently under development, but there is enough to do simple things at the moment. These functions are unoptimised, and run rather slowly for large amounts of geometry. I don't plan to support any different file formats, but let me know if there are others which would be useful.

OBJ format support

Wavefront OBJ files are used to represent single objects. They can be loaded or saved from fluxus, and read or written from other applications. See the documentation for the commands:

```
(obj-import filename)
(obj-export filename obj-list type)
```

COLLADA format support

Collada is a standard file format for complex 3D scenes. Collada files can be loaded, currently supported geometry is triangular data, vertex positions, normals and texture coordinates. The plan is to use collada for complex scenes containing different geometry types, including animation and physics data. Collada export is planned.

See the documentation for the command:

```
(collada-import filename)
```

0.12 Shaders

Hardware shaders allow you to have much finer control over the graphics pipeline used to display your objects. Fluxus has commands to set and control GLSL shaders from your scheme scripts, and even edit your shaders in the fluxus editor. GLSL is the OpenGL standard for shaders across various graphics card types, if your card and driver support OpenGL2, this should work for you.

```
(shader vertshader fragshader)
```

Loads, compiles and binds the vertex and fragment shaders on to current state or grabbed primitive.

```
(shader-set! paramlist)
```

Sets uniform parameters for the shader in a token, value list, eg:

```
(list "specular" 0.5 "mycolour" (vector 1 0 0))
```

This is very simple to set up - in your GLSL shader you just need to declare a uniform value eg:

```
uniform float deformamount;
```

This is then set by calling from scheme:

```
(shader-set! (list "deformamount" 1.4))
```

The deformamount is set once per object/shader - hence it's a uniform value across the whole object.

Shaders also get given all pdata as attribute (per vertex) parameters, so you can share all this information between shaders and scripts in a similar way:

In GLSL:

```
attribute vec3 testcol;
```

To pass this from scheme, first create some new pdata with a matching name:

```
(pdata-add "testcol" "v")
```

Then you can set it in the same way as any other pdata, controlling shader parameters on a per-vertex basis.

Samplers

Samplers are the hardware shading word for textures, the word sampler is used to be a little more general in that they are used as a way of passing lots of information (which may not be visual in nature) around between shaders. Passing textures into GLSL shaders from fluxus is again fairly simple:

In your GLSL shader:

```
uniform sampler2D mytexture;
```

In scheme:

```
(texture (load-texture "mytexturefile.png"))
    (shader-set! (list "mytexture" 0))
```

This just tells GLSL to use the first texture unit (0) as the sampler for mytexture. This is the texture unit that the standard (texture) command loads textures to.

To pass more than one texture, you need multitexturing turned on:

In GLSL:

```
   uniform sampler2D mytexture;
uniform sampler2D mysecondtexture;
```

In scheme:

```
(multitexture 0 (load-texture "mytexturefile.png"))  ; load to texture unit 0
    (multitexture 1 (load-texture "mytexturefile2.png")) ; load to texture unit 1
    (shader-set! (list "mytexture" 0 "mysecondtexture" 1))
```

0.13 Turtle Builder

The turtle polybuilder is an experimental way of building polygonal objects using a logo style turtle in 3D space. As you drive the turtle around you can place vertices and build shapes procedurally. The turtle can also be used to deform existing polygonal primitives, by attaching it to objects you have already created.

This script simply builds a single polygon circle, by playing the age old turtle trick of looping a function that moves a bit, turns a bit...

A circle

```
(define (build n)
    (turtle-reset)
    (turtle-prim 4)
    (build-loop n n)
    (turtle-build))

(define (build-loop n t)
    (turtle-turn (vector 0 (/ 360 t) 0))
    (turtle-move 1)
    (turtle-vert)
    (if (< n 1)
        0
        (build-loop (- n 1) t)))


(backfacecull 0)
(clear)
(hint-unlit)
(hint-wire)
(line-width 4)
(build 10)
```

For a more complex example, just modify the (build-loop) function as so:

A circle of circles

```
(define (build-loop n t)
    (turtle-turn (vector 0 (/ 360 t) 0))
    (turtle-move 1)
    (turtle-vert)
```

```
(if (< n 1)
    0
    (begin
        (build-loop (- n 1) t)          ; add another call to the recursion
        (turtle-turn (vector 0 0 45))   ; twist a bit
        (build-loop (- n 1) t))))
```

0.14 Making Movies

Fluxus is designed for realtime use, this means interactive performance or games mainly, but you can also use the frame dump commands to save out frames which can be converted to movies. This process can be fairly complex, if you want to sync visuals to audio, osc or keyboard input.

Used alone, frame dumping will simply save out frames as fast as your machine can render and save them to disk. This is useful in some cases, but not if we want to create a movie at a fixed framerate, but with the same timing as they are generated at - ie synced with an audio track at 25fps.

Syncing to audio

The (process) command does several things, it switches the audio from the jack input source to a file, but it also makes sure that every buffer of audio is used to produce exactly one frame. Usually in realtime operation, audio buffers will be skipped or duplicated, depending on the variable framerate and fixed audio rate.

So, what this actually means is that if we want to produce video at 25fps, with audio at 44100 samplerate, 44100/25 = 1764 audio samples per frame. Set your (start-audio) buffer setting to this size. Then all you need to do is make sure the calls to (process) and (start-framedump) happen on the same frame, so that the first frame is at the start of the audio. As this process is not realtime, you can set your resolution as large as you want, or make the script as complex as you like.

Syncing with keyboard input for livecoding recordings

You can use the keypress recorder to save livecoding performances and rerender them later.

To use the key press recorder, start fluxus with -r or -p (see in-fluxus docs for more info). It records the timing of each keypress to a file, it can then replay them at different framerates correctly.

The keypress recorder works with the process command in the same way as the audio does (you always need an audio track, even if it's silence). So the recorder will advance the number of seconds per frame as it renders, rather than using the realtime clock - so again, you can make the rendering as slow as you like, it will appear correct when you view the movie.

Recording OSC messages is also possible (for storing things like gamepad activity). Let me know if you want to do this.

Syncing Problems Troubleshooting

Getting the syncing right when combining audio input can be a bit tricky. Some common problems I've seen with the resulting movies fall into two categories.

Syncing lags, and gets worse with time

The call to (start-audio) has the wrong buffer size. As I set this in my .fluxus.scm I often forget this. Set it correctly and re-render. Some lagging may happen unavoidably with really long (over 20 minutes or so) animations.

Syncing is offset in a constant manner

This happens when the start of the audio does not quite match the first frame. You can try adding or removing some silence at the beginning of the audio track to sort this out. I often just encode the first couple of seconds until I get it right.

0.15 Fluxus In DrScheme

DrScheme is a "integrated development environment" for Scheme, and it comes as part of PLT scheme, so you will have it installed if you are using fluxus.

You can use it instead of the built in fluxus scratchpad editor for writing scheme scripts. Reasons to want to do this include:

- The ability to profile and debug scheme code
- Better editing environment than the fluxus scratchpad editor will ever provide
- Makes fluxus useful in more ways (with the addition of widgets, multiple views etc)

I use it a lot for writing larger scheme scripts. All you need to do is add the following line to the top of your fluxus script:

```
(require (lib "drflux.ss" "fluxus-version"))
```

Where version is the current version number, eg "fluxus-0.14". Load the script into DrScheme and press F5 to run it - a new window should pop up, displaying your graphics as normal. Rerunning the script in DrScheme should update the graphics window automatically.

Known issues

Some commands are known to crash DrScheme, (show-fps) should not be used. Hardware shading probably won't work. Also DrScheme requires a lot of memory, which can cause problems.

0.16 Fluxus Scratchpad And Modules

This chapter documents fluxus in slightly lower level, only required if you want to hack a bit more :)

Fluxus consists of two halves. One half is the window containing a script editor rendered on top of the scene display render. This is called the fluxus scratchpad, and it's the way to use fluxus for livecoding and general playing.

The other half is the modules which provide functions for doing graphics, these can be loaded into any mzscheme interpreter, and run in any OpenGL context.

Modules

Fluxus's functionality is further split between different Scheme modules. You don't need to know any of this to simply use fluxus as is, as they are all loaded and setup for you.

fluxus-engine

This binary extension contains the core rendering functions, and the majority of the commands.

fluxus-audio

A binary extension containing a jack client and fft processor commands.

fluxus-osc

A binary extension with the osc server and client, and message commands.

scheme modules

There are also many scheme modules which come with fluxus. Some of these form the scratchpad interface and give you mouse/key input and camera setup, others are layers on top of the fluxus-engine to make it more convenient. This is where things like the with-* and pdata-map! macros are specified and the import/export for example.
