Gem

Graphics Environment for Multimedia

IOhannes m zmölnig
forum::für::umläute

April 10, 2003

Contents

1 How to read this document

This document intends to help making some basic steps into **Gem**. It was created because of the need for some structure of several workshops about **Gem**, which i was dearly missing – resulting in (i guess) much confusion on the participants' side about what was going on.

I assume that the reader of this text has a fairly recent version of **pd** already running on her computer. Furthermore I assume, that she is already familiar with the basic concepts of building patches with **pd**, with the differences between object- and message-boxes and with the differences between messages and signals.

1.1 Transcribing pd-patches

Since **pd** is a graphical programming language it is often hard to explain patches within text-documents. Therefore I will try to to give a lot of screenshots of working pd-patches, which can be used to reproduce things. However, to refer to **pd**-objects within text, i will use following typesetting-conventions:

`[foo 1 2]`	object (with arguments)	fixed-font within square brackets
`[foo` *x*`]`	object (substitute arguments)	
`[foo bar(`	message	fixed-font within bracket/paranthese
`[foo` *name*`(`	message (with substitution)	

The term "*connecting object* `[foo]` *to object* `[bar]`" means, *connecting the first outlet of* `[foo]` *(that's the leftmost little freckle on the lower edge of the object) to the first inlet of* `[bar]` *(the leftmost little freckle on the upper edge of the object)*. It does not mean, connecting any other outlet of `[foo]` to any other inlet of `[bar]` (this would be stated explicitely). And it does not mean, connecting *any* outlet of `[bar]` to *any* inlet of `[foo]`. **pd** only allows connecting outlets with inlets (not the other way round), thus you really have to start at object `[foo]` and draw a line to object `[bar]`.

2 openGL

`TODO`: description of 3D-scenes, hardware acceleration,. . .

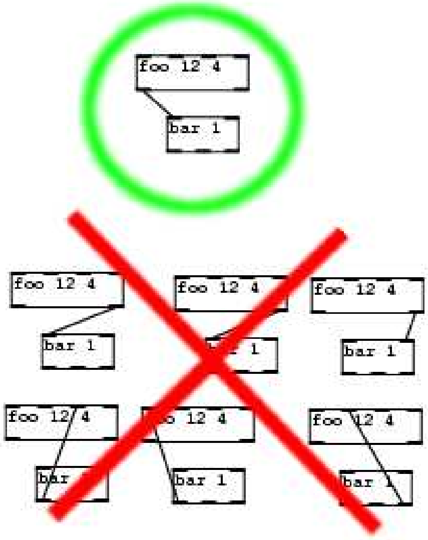

Figure 1: connecting `[foo]` to `[bar]`

2.1 meters, inches, fathoms,...

Working with computer graphics it is quite common that the size of a graphical element is given in *pixels*. The size of a *pixel* depends on your hardware (monitor-resolution). This is fine, when working with bitmap-graphics (because bitmaps are based on pixels). However, if you are drawing an object in a 3D-application, the actual size (on the screen) of this object will heavily depend on the distance between the eye/camera and the object (objects farther away being smaller), although the logical size of the object is constant. Since vector-based applications (like **Gem**) describe objects in a logical way, they do not store sizes in pixels (as *units on the screen*). Instead, lengths are given in measures without any dimension. If a line is “1” unit long, you do not know, whether this is one meter, one AU or one fathom. But you can be sure, that logically this line is half as long as a line with a length of “2”.

3 A first Gem-patch

Gem-patches are pd-patches. They only use a special set of objects that are not native to pd but are provided by Gem. These objects can be connected to "normal" pd-objects, either being controlled by them or controlling them. However, Gem-objects form a coherent unit: chances are, that you have to use *several Gem-objects together* to get the desired result. Normally, it does **not** make sense to use a single, isolated Gem-object: they are not standalone objects.

3.1 `[gemwin]`

There is one object that really ***must*** exist in order to make Gem work . This is `[gemwin]`.

`[gemwin]` is a handle to a *display context*. This basically means, that it provides a window (thus the name), where the Gem-scene is rendered to, the "Gem-window".

To create the Gem-window, you have to send a `[create(` message to `[gemwin]`. To destroy this window, just send a `[destroy(` message.

As you might notice, that even though a Gem-window (named "Gem") is created, nothing is drawn to this window. Instead it will initially contain what is *behind* the window (which is often quite irritating). Thus it holds literally "nothing", not even "black".

To display something on the Gem-window, you have to turn on rendering. This can be done, by simply sending a `|1(` to the `[gemwin]`. To turn off rendering again, just send a `|0(`.

This most simple Gem-patch I can think of is shown in figure 2. It will display a black window, as no objects are added to the Gem-scene.

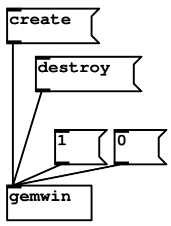

Figure 2: the mandatory part of each Gem-patch

You can only turn rendering on, when there is an already created Gem-window. If you destroy the Gem-window while rendering, rendering is turned off automatically.

If you want to change the properties of the Gem-window, you can do so by sending messages to the `[gemwin]`. For instance you could change the title that is displayed in the title-bar, by sending `[title` *`title-symbol`*`(`.

You can also set the size (`[dimen` *`width height`*`(`) and position (`[offset` *`x y`*`(`) of the Gem-window. Note that you have to send messages concerning the window-appearance ***before*** actually creating the Gem-window. You cannot resize or move an existing Gem-window with messages.

Note also, that it is not possible to resize the Gem-window under *linux* by simply using the window-manager, although this is possible under *windos*.

If you want to have a *fullscreen* output of Gem, you can send it a `[fullscreen 1(` (before creating the "window"). Note that since this uses all the screen, you might not be able to control your underlying pd any longer, for instance you might not be able to close the Gem-window again (and leave fullscreen-mode). You can disable fullscreen-mode by sending a `[fullscreen 0(`.

Until now, only one Gem-window can be used at one time (within the context of one running instance of pd). You can put as many `[gemwin]`s into your patch(es), but they all will affect the same Gem-window.

future-music: It is planned to support multiple independent Gem-windows in future releases of Gem. These will not necessarily be windows on your desktop, but could be the TV-output of your video-card, an IEEE1394-link or a connection to a streaming-video-server to broadcast over the internet.

3.2 [gemhead]

While `[gemwin]` creates a window where the rendering is done, `[gemhead]` is the object that connects a set of Gem-objects to this rendering context.

When doing audio-processing with pd, you have to connect all the signal-objects you want to "hear" to the `[dac~ ]` somehow. All the Gem-objects have to be connected to the `[gemhead]`. Contrary to the `[dac~ ]`, the `[gemhead]` is not the last object (the sink, where the final signal goes), but the **first** object (the source, where the Gem-list starts).

Each rendering cycle (each time a frame is drawn – normally at a rate of 20 fps) the `[gemhead]` triggers an output, the so called *Gem-list*. This Gem-list is a special kind of message, that is understood only by Gem-objects.

> *Generally, a Gem-object will "work"* (eg: affect what is drawn to the Gem-window) *if and only if a Gem-list is sent to its first (leftmost) inlet.*

Very few Gem-objects have more (two) inlets for Gem-lists.

To stack several Gem-objects (to let their effects sum up), each Gem-object has at least one (leftmost) outlet, that passes through the Gem-list.

3.3 Let's draw something

The most basic Gem-objects are probably the so called *Geos*, which are "primitive" shapes. These are *drawables* that tell the renderer to "draw a certain shape". One of these drawables is `[square]`, that will (surprisingly) draw a square.

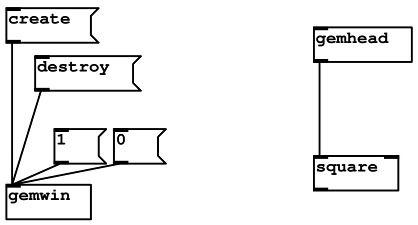

Figure 3: drawing a square

You will notice, that the `[square]` has 2 inlets. The first one if for the Gem-list. The remaining one can be used to set parameters. The (only) parameter that can be set for `[square]` is the length of it's edge. You can set (and change) it by sending a number to the second inlet. Initially this is "1", unless this is overridden via a first argument, like in `[square 2.1]`.

There are a lot of primitive Geos, like `[rectangle` *`a b`*`]`, `[triangle` *`edge`*`]`, `[cube` *`edge`*`]`, `[circle` *`diameter slices`*`]`, `[disk` *`outerradius slices innerradius`*`]`, `[cylinder` *`diameter=height]`*, `[cone` *`diameter=height]`*, `[sphere` *`diameter]`*.

3.4 Colouring a Geo

It is a fine thing to have Geos, but it soon becomes boring, if all you can do is change their size. To do more sophisticated things, there are so called *Control*-objects, that allow Geos to be moved, coloured,. . .

Generally, control-objects are called **before** Geos. This is because of the openGL-architecture as a state machine.

Control-objects tell the renderer to change the appropriate state. For instance, the `[color]` object will tell the renderer "to take the brush with color *x*". Everything that is drawn afterwards will be coloured with *x*, until another `[color]` object resets the colour of the renderer.

You have to pass the colour you want for painting to `[color]` as sets of Red/Green/Blue values, each ranging between 0 and 1. This can be done by giving initial arguments: `[color` *`red green blue`*`]` If you don't specify any arguments, `1 1 1` is assumed, indicating "white". To change the colour during runtime, you can send a list of the values (an rgb-triplet) to the second inlet of `[color]`.

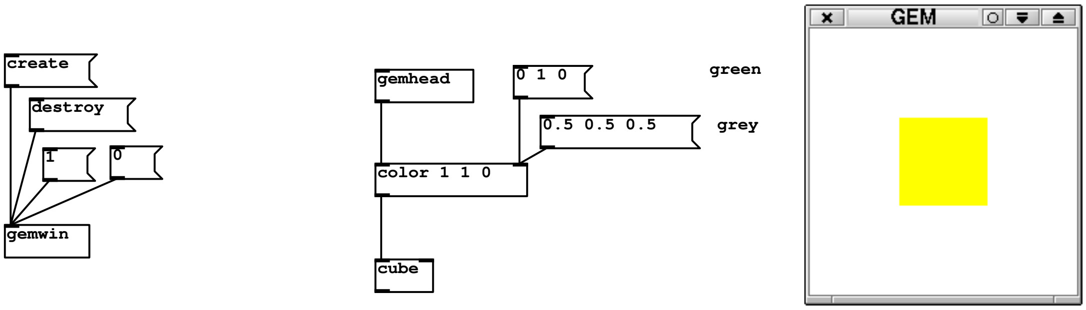

Figure 4: colouring a Geo

An alternative is `[colorRGB]` which allows to set the values for each colour-component separately via float-inlets.

3.5 moving around

When connecting a Geo to `[gemhead]`, it is drawn in the middle of the screen. Each Geo has it's *pivot point* which is the centroid of the shape for simple shapes. (And a rather arbitrary but hopefully intuitive point for more complex shapes). When the renderer is told to draw a Geo, it will place the pivot-point of the Geo on the origin of the ordinates of the 3D-world. The "camera" that takes the pictures that are the display in the Gem-window, is aligned to focus on the "world-origin" of the 3D-scene. Thus all the Geos appear in the center of the screen. We can move around the objects (the Geos, not the pd-objects) with `[translate]` resp. `[translateXYZ]`.

[translateXYZ *transX transY transZ*] takes 3 arguments as the initial translation in the direction of the 3 axis X (from left to right), Y (from below to up) and Z (from in front of the screen into the screen). The partial translations can also be set via numbers to the 3 right inlets.

[translate *factor vecX vecY vecZ*] the direction-vector *vecX vecY vecZ* is scaled by *factor* to get the translation. Thus you can specify the direction of the translation (third inlet) indendently from the amount of translation (second inlet).

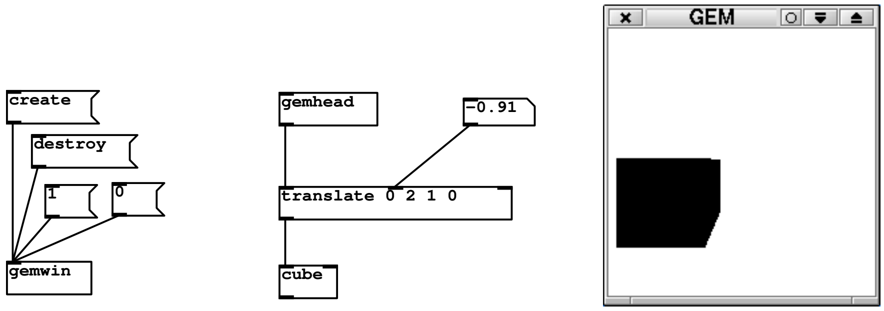

Figure 5: translation of a Geo

3.6 turning around

Any Geo can be rotated around it's pivot point.

[rotateXYZ *rotX rotY rotZ*] rotates the Geo, firstly around the X-axis by *rotX*°, then around the Y-axis by *rotY*° and finally around the Z-axis by *rotZ*°.

[rotateXYZ *amount vecX vecY vecZ*] rotates the Geo, by *amount*° around the vector specified by the direction-vector *vecX vecY vecZ*.

3.7 local coordinate system vs. world coordinate system

[translate] ([translateXYZ]) resp. [rotate] ([rotateXYZ]) are applied to the so called *local coordinate system* rather than the *world coordinate system*. A *translate* translates the local coordinate system by a certain amount, a *rotate* rotates the whole coordinate system. This means that the order of stacked *translate* and *rotate* transformations is **very important**.

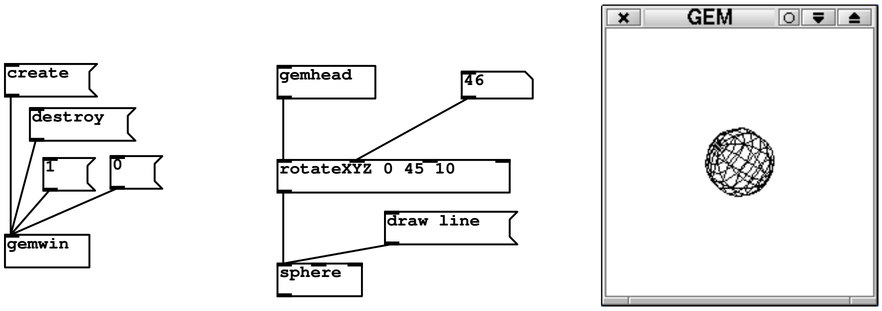

Figure 6: rotating a Geo

Example: Assume a cube, that is translated first and afterwards rotated. If the rotation amount is now changed, the result will be a cube that is rotating around it's (pivot-)axis at a fixed position. If the transformations are swapped (rotating first and afterwards translating), changing the rotation amount will result in a cube, that is rotating around the origin of the scene. (see patch at fig.7 and Gem-output at fig.8)

This behaviour becomes easily understandable when considering "everyday local coordinate systems". For instance, assume that you are standing at a spot X looking straight forward. Your destination place will then heavily depend on whether you turn right by 90° and walk 10 steps ahead or walk 10 steps ahead first and then turn right by 90°.

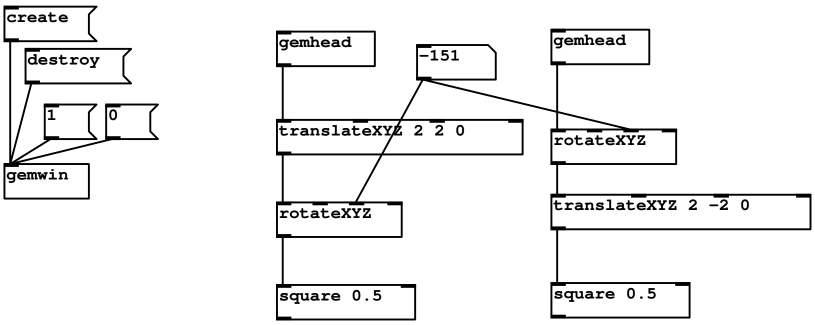

Figure 7: the importance of transformation-ordering

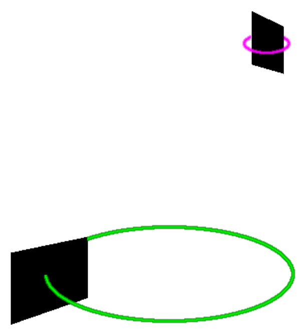

Figure 8: *translate* before *rotate* results in a Geo that rotates around it's own axis (lilac). *rotate* before *translate* produces a Geo that rotates around the origin. (green)

3.8 multiple objects

You can put multiple Geos into a gemlist. When doing transformations (like `[rotate]`) in this gemlist, each object that is "below" this transformation will be affected by it. Objects that are "above" the transformation, will not be affected by it.

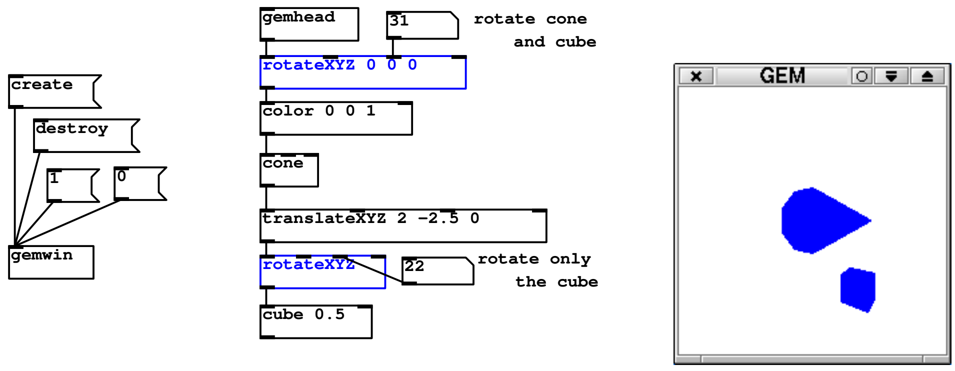

Figure 9: multiple Geos in one gemlist

This is often very convenient, as it allows a hierarchic description of 3D-scenes. Objects that are at the beginning of the gemlist are higher in hierarchy than objects below them. Lower object are considered to be "parts" of higher objects – like your fingers are part of your hand: if you move your hand, the fingers are also moved.

3.9 independent objects

Sometimes you want to control Geos independently of each other. Then you might not want to put them into the same gemlist. Just start a new gemlist with another [gemhead]. Gem-objects that are "childs" (below) different [gemhead]s, do not interfere with each other. (see fig.10)

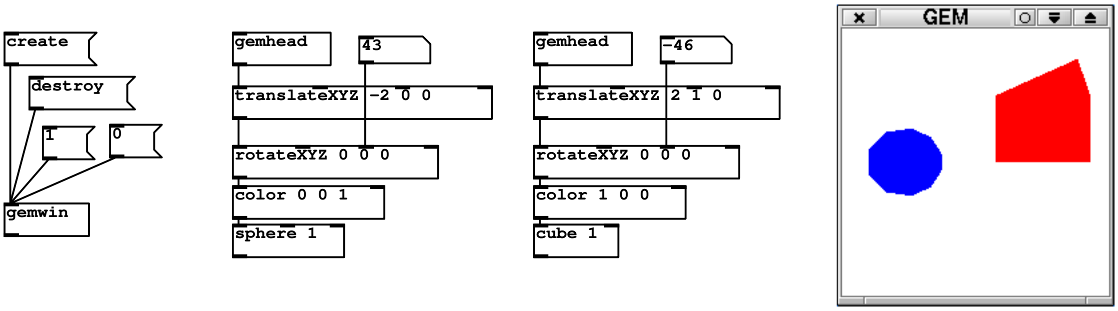

Figure 10: Geos in multiple gemlists

3.10 partly independent objects in a single gemlist

Sometimes you want partly independent control over Gem-objects. For instance if you wanted to model a human body, you might want to move and turn the whole body (with all parts like the head being transformed too) *and* want to be able to turn the head and rise the arm independently. To split a gemlist into independent sub-gemlists you can use the object [separator].

Everything below the [separator] is decoupled from other sub-gemlists under other [separator]s. (see fig.12)

3.11 Prettifying the output

As you might have seen the Geos look quite ugly since you cannot distinguish between different sides of an object. To prettify the rendering we can add some lighting of the scene.

To make use of lighting, we have to put some light-sources into our scene first.

[world_light] is used as an infinitely far away light source (like the sun). Thus you can only specify the direction of the light by rotating the [world_light].

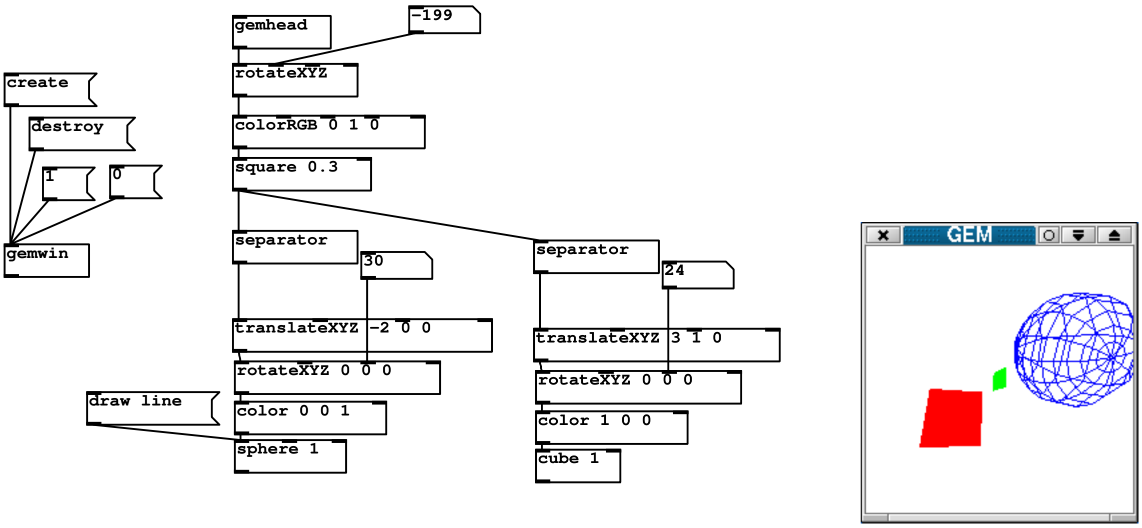

Figure 11: partly independent Geos in a single gemlist

`[light]` is a point-source that can be placed somewhere via translation and rotation.

For positioning a light-source you can send a `[debug 1(` to the light-source, which will display a small cone that represents the light-source.

turning on/off the light: Positioning light-sources is not enough to turn lighting on. Instead you have to turn it on (or off) explicitely. This can be done by sending a `[lighting 1(` (or `[lighting 0(`) to the `[gemwin]`. *Note* that when turning lighting on without having any light-sources, your scene will pitch-black.

3.12 *Animation*

Gem itself has no possibilities to animate things. However, you can control the appearance of each frame with pd-logic.

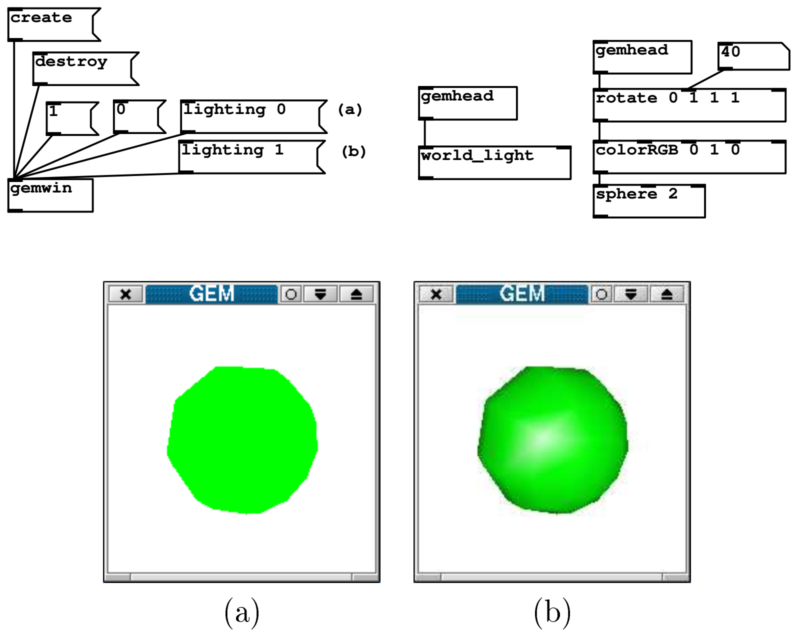

Figure 12: lighting a scene: (a) unlit, (b) lit

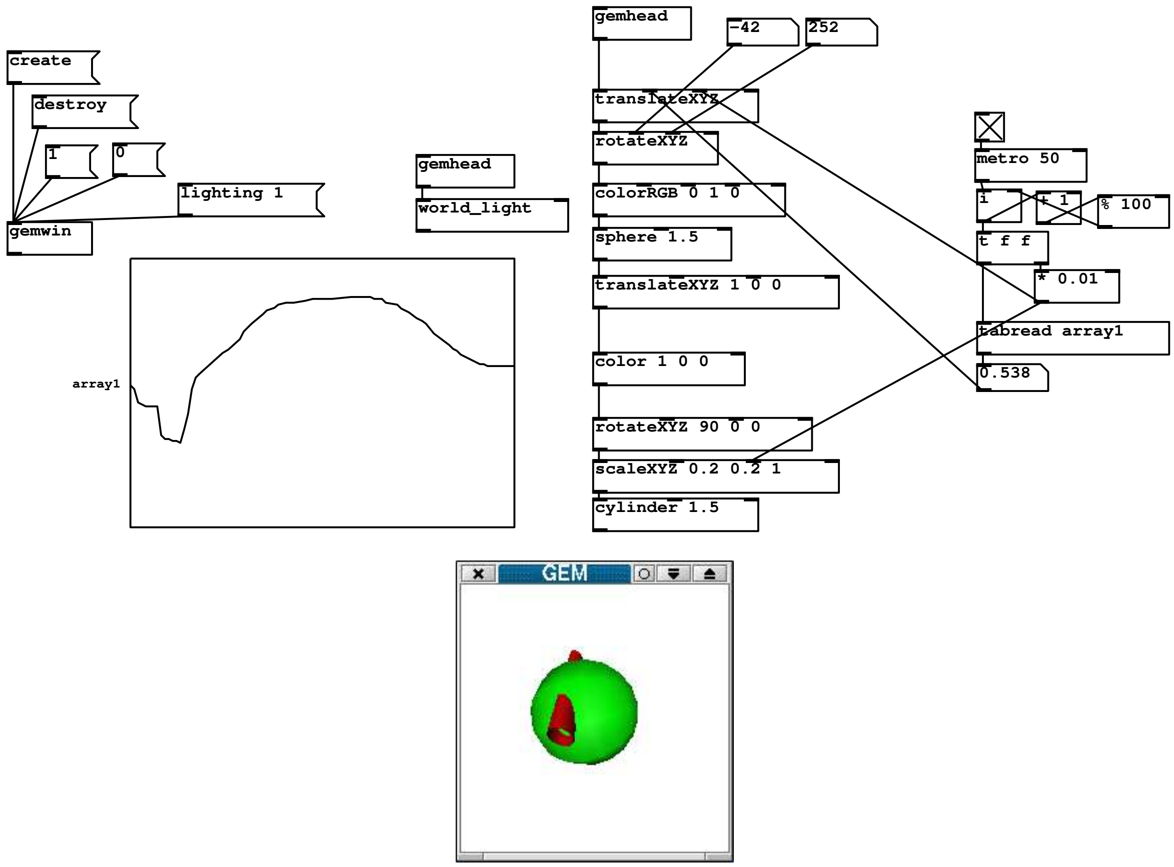

Figure 13: animating things in Gem

3.13 *Exercise*

Do some exercise: Build a model of the solar systems with following astronomic atoms:

- Sol
- Mercury
- Venus
- Terra
- Luna

Assume arbitrary ("good-looking") sizes of planets/moons/stars. Use circular paths. Animate the whole thing (with arbitrary speeds).

4 Images

To display images, Gem has several objects that can be used as "image sources". These include image-loaders for JPEG/TIFF, movie-loaders and live-video sources (camera input). Generally all image-related object start with "`[pix_`" followed by something that describes the functionality.

`[pix_image]` is the simplest image-source. It will load an image into RAM. You can specify the filename of the image either as a first argument, or by sending a `[open` *`filename`*`(` to the object.

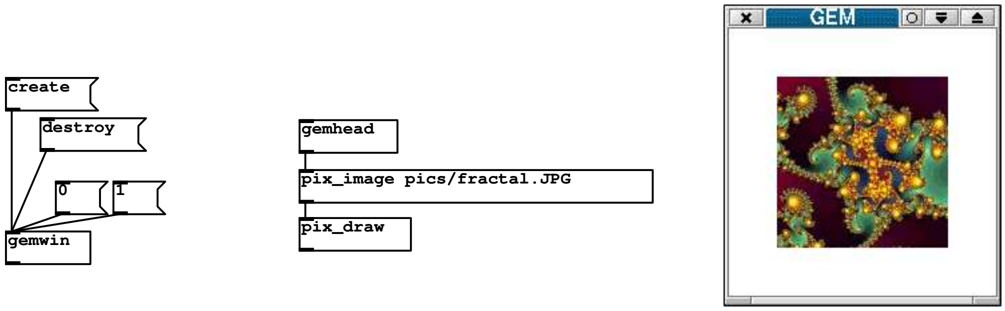

Figure 14: drawing an image

The simplest way to display an image is using `[pix_draw]`, which will plot the image in it's original size centered onto the Gem-window. If the image is bigger than then the Gem-window, it won't be displayed completely. `[pix_draw]` is normally not supported by hardware-accelerated graphics-cards (which makes it very slow generally) and i *highly* recommend not to use it. It is just the easiest way to draw an image onto the screen, that's why it is used here.

4.1 texturing

GEM is an openGL-application. openGL-applications normally do not draw images directly (like with `[pix_draw]`). Instead images are used as *textures* on the Geos. You can think of texturing as putting a wallpaper on the Geo.

`[pix_texture]` is used for applying a texture. This means that the image is uploaded onto the memory of the graphics-card. Each Geo that is drawn after uploading the texture in the same gemlist will have this texture applied.

Note: hopefully [pix_texture] and [pix_texture2] will merge sometime.

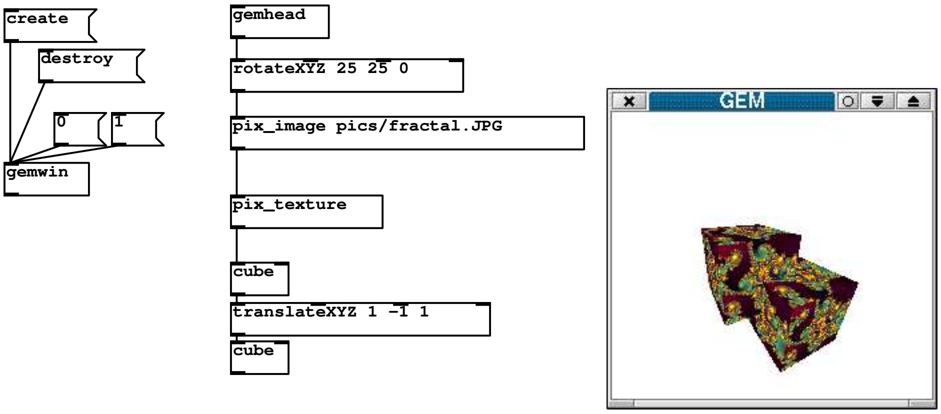

Figure 15: texturing an image

Unfortunately openGL only supports textures the width (and height) of which are powers of 2 (like 512x256 pixels). If you try to apply a "wrong sized" image as a texture with [pix_texture] you will not see anything (like not-texturing at all).

To overcome this, there is the [pix_texture2] object, which allows images of any size (like 320x240 pixels) to be textured on Geos.[1]

[1] on macOS-X any Geos are supported, on linux and windows you might experience weird results with "complex" Geos (like [sphere]).

5 *Solutions for Exercises*

5.1 Solar System (3.13)