08/01/12 Pd User's gathering at Shin-Osaka, Japan
Presentation material on GEM (prepared by David Shimamoto <gyokimae at h o t m a i l d o t c o m>)

Intro

This document is a based on study notes taken by the author. Due to its nature, the accuracy of its contents is not to guaranteed.

Overview

Gem is perhaps the most widely used graphic extension library for Pd. Such features as the following are available.

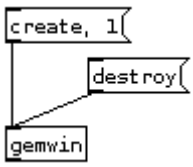
- Render 3D graphics
- Load and use 3D models in .obj format.
- Easy handling of texture mapping. Both still images and video can be used as source material.
- Handle input from mouse & keyboards
- Manage live video input. Also easily handle elementary motion detection.
- Color conversion and various keying of video (such as 'blue back' and else)

Some background on GEM.

Many of the functions implemented in GEM are wrappers to 'OpenGL' which is a popular library to handle 3D graphics. If you ever question the purpose of certain features, it is often likely that it is there because that is the way OpenGL is designed, such flags already exists.
Also, for such reasons it is highly recommended to have a Video card supporting OpenGL when working with GEM.

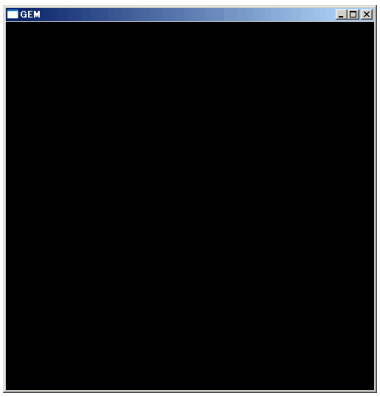
Basics of GEM(1) – Preparing a “Gem Window”

All features of GEM will be rendered on a windows separate from what Pd provides. Therefore, we must first prepare a window to work in. The procedures are easy. As illustrated below, create a [gemwin] object and send it a [create< message. This only creates the window. In order to activate the rendering, you then need to send a [1< message in addition. An option is to also send both with a comma to separate the commands. The [gemwin] object knows how to interpret this properly.

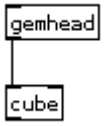


To close a GEM window, you can either send a [destroy< message or simply delete the [gemwin] object. Please refer to the help file to [gemwin] for details on available parameters.

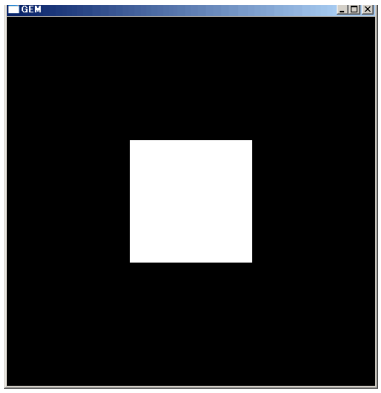
Once your GEM windows is ready, you are mostly done using these objects. Feel free to leave them alone in any part of your patch.

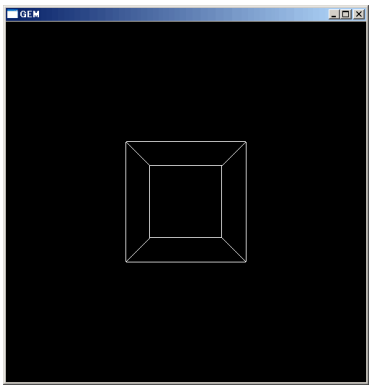


Basics of GEM(2) – Drawing Geos



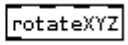
We will now draw a cube in the created window.
The first thing we will need to do is create a [gemhead] object. This object outputs a literal 'command' to render something on the GEM window at frame rate. When connecting this output to a [cube] object which knows how to manage this command, you will see a cube appear in the window (Image in the lower left)
At the moment, this looks more like a plain rectangle than a 3-dimensional cube. The lower right image is only the frame of the cube displayed. Please refer to the help file on [cube] on how to render objects in different behaviors)



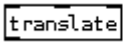


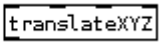
What you have seen so far are the fundamental concepts of managing 3D objects in GEM.
To transform the object, you will insert objects which add detail to the 'command'.

Below are some major objects used for transformation.

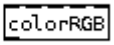


Add rotation.





Add translation (position)



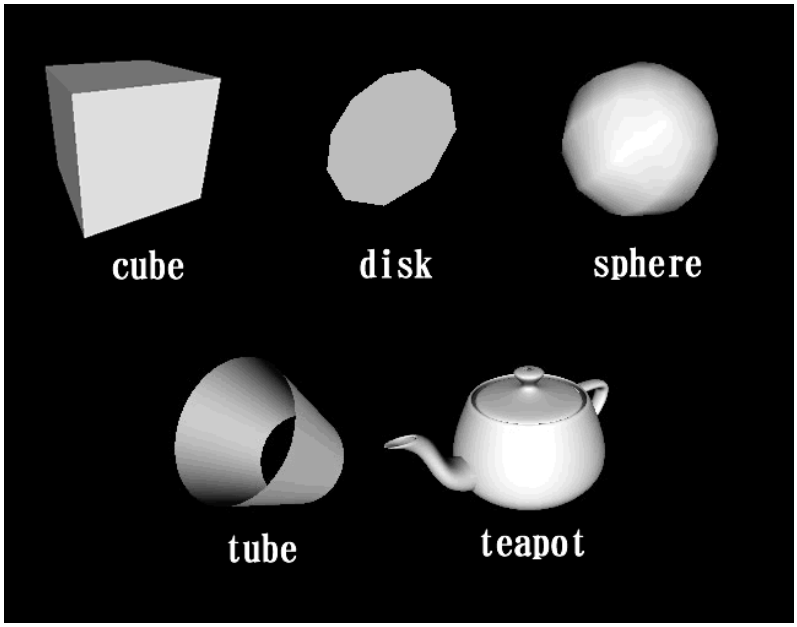
Add color information.

Some tips on using [gemhead] (intermediate)

As mentioned, [gemhead] outputs messages at 'frame rate' (default is 20fps). This means that for example, by connecting a [bang< message to the output of [gemhead] you get a chain which generates 'bang' messages at frame rate. Another example is to connect a [repeat] object to the output of [gemhead]. This enables you to draw multiple objects of the same type using a single pair of [gemhead] and geo objects such as [cube]. However, this may not be apparent without additional transformation techniques since all of the objects will appear with the same parameters over lapping at the same position.

Primitive geos.

GEM provides several objects to draw elementary shapes. There are also alternative ways of drawing objects in 3D space, such as specifying all points, or simply load a 3D model saved in a .obj format file. obj files can be created using external applications referred to as '3D modeling' tools. 'Blender' is one of the preferred free software available.



So far we have studied basic concepts on manging geos in GEM. This is all I am mentioning for now on how to manipulate objects as there are many useful 'step by step' tutorials online. Here is a quick review on the only three concepts I expect for you grasp for now.

- The [gemhead] is the important object at the top of the chain. It creates a rendering command at frame rate.
- You can place objects in the 3D space by sending this 'command' to objects which can interpret it and execute.
- The behavior of the rendered object such as rotation and position are manipulated by inserting objects in between the basic chain, adding additional information to the 'command'.

Extra note.
All objects in the image above are rotated from the default position.
Also, you will probably find the objects painted in a simplex color at default. In order to add shades and a since of lighting effect, you will need to enable the 'lighting' feature by sending the appropriate message to the [gemwin] object we initially created. You can then place a [world_light] object to add lighting from a desired direction. Please refer to manuals on [gemwin] and [world_light] for detail].

In case you wondered, this flag to enable/disable the lighting and its defaulting to off are all behaviors of the OpenGL API mentioned earlier.
Also, don't be surprised if your window darkens out when enabling the lighting feature. This is the obvious result if you have not yet placed any lighting source in your image!

About particles

Particles are, as you may know, a terminology taken from chemistry. In computer graphics it is a popular technique used when grouping hundreds and thousands of objects moving autonomic, but under a certain discipline. It is often used for purposes such as

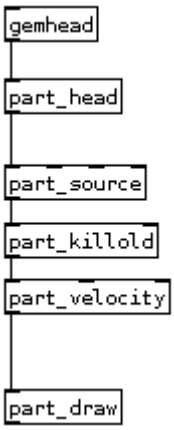
- Burning flames and smoke arising.
- Many leaves or snow flakes falling
- Flickering partials of light.

As is GEM uses the OpenGL library for its rendering, an open API named 'Particle System API' is used to manage the particles.

Side tip:
The 'Particle System API' has been increasing its version and now has some features added not covered by the version in GEM. However the authors of GEM have rewritten much of the code in the particle API (opposed to using it as an external library) which I imagine is making it difficult to implement new features provided by the API.

Basic connection for using particles

There is a set pattern of chinning objects which is used for almost all situations. It is nice to get acquainted with this pattern which I will introduce below.
Please keep in mind that this is only to demonstrate the basic concept and flow. Many of these objects will either not work at all or have unexpected manners until sending them arguments.



Let's take a look from top to bottom.
As you would with rendering any other object in the GEM window, you will need to begin the chain with a [gemhead] object. This will be connected to a [part_head] object which manages the group of particles.

We will now need a [part_source] object. This executes the process of generating new particles in 3D space.
[part_killold] eliminates particles that have existed longer than the specified time. Without this object, the number of particles existing in space may continue to increase to an infinite number . Well, actually there is an argument controlling the maximum number of particles that may exists. However, exceeding this number does not harm to your application but may result in unpredictable motion.
The key is to pair [part_source] to adjust the rate of particles appearing and [part_killold] to control its mass in order to archive the desired effect.

The next thing you will probably want to look into is giving initial velocity (a direction and speed of movement) to the particles. Else, you will find a blob of particles which continue to appear and then stay still until it parishes at the same position.

In the example above, we have placed a [part_draw] object to draw each of the particles as points in the window. GEM also provides ways of rendering each particles as geos in which case, you will find a field full of cubes, spheres or any shape of your choice flying around.

Note added after workshop:
As I lously demonstrated, as a matter of fact the particles do seem to have a default velocity. When bypassing the [part_velocity] the particles will spread as if it were given a 'sphere' as a default domain. This was cool and 'wowed most of the audits but not exactly what I had expected to happen.

Note :
The maximum number of particles that may exists in the 3D space is set inside the [part_head] object. This defaults to one thousand and can be modified. When the number of existing particles try to exists this value, the [part_source] object simply stops generating new particles.

What happens then? The [part_source] object stands by until some particles parish and then generate new ones when possible. This results in a effect where the source of particles sequentially turn itself on and off.
In order to avoid this, you must set the maximum number of particles available to a reasonable figure and set [part_source] and methods of perishing them so that the number of existing particles do not reach the maximum.

About domains.

So far, we have only seen particles appear in a specific space. Now, how can we expand the region where the particles appear in?

GEM, and the the particle API provides a concept they refer to as 'Domains' which is a method of easily specifying a certain area in the 3D space.

Below are some of the objects you can specify domains along with its effect.

[part_source] Newly created particles will appear in a random position within the domain.

[part_velocity] A newly created particle will be given an initial velocity directing to a random point within the domain. The speed will also vary according to how far the point is.

[part_sink] Particles exceeding (moving out of) the specified domain will instantly parish.

Available types of domains.

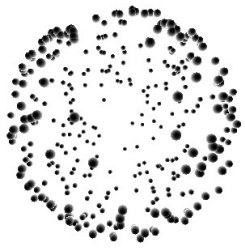
The rest of the document will list the domains and the parameters each accepts. Since the domains are generally invisible, I have applied them to [part_source] to generate the particles within its range to visualize its approximate area. Each particle within the domain is given a fixed size so that you can tell smaller ones are farther away.

Each domain accepts different number of arguments, ranging from 3 to 9.

'point'
A point in space
arg.1 X pos
arg.2 Y pos
arg.3 Z pos

'line'
A line between two points.
arg.1-3 Positions X, Y, Z of point 0.
arg.4-6 Positions X, Y, Z of point 1.

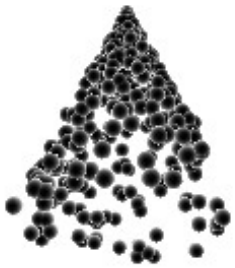
'rectangle'
A rectangular space between four points.
The fourth point is assumed automatically.
arg.1-3 Positions X, Y, Z of point 0.
arg.4-6 Positions X, Y, Z of point 1.
arg.7-9 Positions X, Y, Z of point 2.



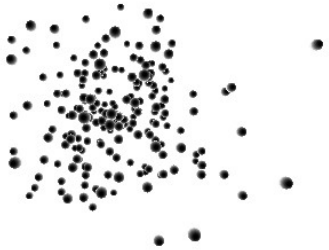
'sphere'
A space within a sphere. When argument 5 is larger than zero, the core is left blank with the given radius.
arg.1-3 Positions X, Y, Z of center.
arg.4 Outer radius
arg.5 Inner radius

'box'
A cube (Sorry for the lousy image)
By specifying the position of two diagonal corners, the rest will be assumed.
arg.1-3 Positions X, Y, Z of point 0.
arg.4-6 Positions X, Y, Z of point 1.

'cylinder'
A cylinder
When argument 8 is larger than zero, the core is left blank with the given radius.
arg.1-3 Positions X, Y, Z of point 0.
arg.4-6 Positions X, Y, Z of point 1.
arg.7 Outer radius
arg.8 Inner radius



'cone'
A cone
When argument 8 is larger than zero, the core is left blank with the given radius.
arg.1-3 Positions X, Y, Z of point 0.
arg.4-6 Positions X, Y, Z of point 1.
arg.7 Outer radius
arg.8 Inner radius



'blob'
Not too sure what this is other than that it seems to be a random range.
arg.1-3 Positions X, Y, Z of center.
arg.4 Standard deviation (??)

'plane'
An Euclid plane.
You will being specifying two points. The plain will include point 0 and perpendicular to a line between the two points.
arg.1-3 Positions X, Y, Z of point 0.
arg.4-6 Positions X, Y, Z of point 1.

Note: When used with [part_source] this domain always return the position of point 0 and is useless for such context.

'disc'
A disc
The concept of arguments 1 through 6 are identical to 'plane'.
You can additionally limit the area by specifying the outer/inner radius of a circle centering at point 0.
arg.1-3 Positions X, Y, Z of point 0.
arg.4-6 Positions X, Y, Z of point 1.
arg.7 Outer radius
arg.8 Inner radius

Some links

OpenGL (http://www.opengl.org/)

Particle Systems API (http://www.particlesystems.org/)

GEM for MaxMSP (http://gem4mac.sourceforge.net/)

Blender (http://www.blender.org/)

Extra notes on rendering particales as geos.

I find two different methods of rendering particles as geos instead of points.

[<your particle chain>]
 |
[part_render]
 |
[cube]

This generally gives except results in your GEM window. However, the console will be flooded with error messages printed at least one per frame.

The second method I am aware of calls for complexity but generates no errors.

[<your particle chain>]
 |
[part_info]
 |
[separator]
 |
[<your translation objects>]
 |
[cube]

[part_info] seems to manage each of the particle as independent chains. There for, the lack of the [separator] will result in unexpected results. (Mainly, multiple particles sharing the same parameters, resulting in overlapping particles appearing as one and a lot of flickering)

Also, [part_info] divides the values such as color and position information to separate outlets of its object. In order to reposition the particles, you will need to prepare your own objects such as [translate] or [color] and feed them outlets 3 and 4 of [part_info]

In other words, you now have the option of preparing an array containing custom values for each particle. You can then give independent colors or rotation angles to each of the particles.

I am not aware of what the proper method is and how these natures mentioned relate to the internals of the program. I appreciate advise from those fluent with this part of the application.