GEM's rendering engine: a mystery unrevealed

Johannes M Zmölnig
Institute of Electronic Music and Acoustics,
University of Music and Dramatic Arts, Graz
zmoelnig@iem.at

Abstract

Gem was the first extension to Miller Puckette's pure-data *that allowed graphics-processing within this environment. Because Gem's hierarchical approach differs significantly from pd's "native" linear signal-flow model, the underlying rendering engine has proven to be a bit complicated. What's more, this engine keeps changing, which makes it even harder to understand. This article aims to shed some light on what is the core part of more than a 100.000 lines of source-code.*

1 Introduction

Gem has been around for almost 10 years. The original *Graphics Environment for Max* has undergone a lot of changes until the current form has been reached. Not only the number of objects has increased, but also the underlying rendering system has changed completely several times. Apart from adding new features, the main reason for such changes has been the problem of integrating a 3D-modelling system within pd's signal-flow structure.

Patcher-languages (like pd) have a very linear processing structure: A signal is generated by a source, it is passed to a modifying object (e.g. a filter), then it is passed to another modifying object (and so on), and finally it is sent to a sink (e.g. the soundcard-output) (see fig.1)

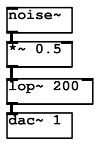

Figure 1: the signal flow in pd

This structure applies very well to video-processing (a video-stream is modified by several "effects" and finally sent to the monitor).

However it does not apply too well to vector-graphics, which are generally best described by a hierarchical model.

Due to the graphical nature of patcher-languages, they seem ideal to describe hierarchical models. However it sometimes needs some tweaking to circumvent their linear nature.

2 An early attempt: using signal-flows (–1996)

The first versions of GEM for the Max-platform tried to utilize the linear signal-flow model for doing graphics processing (see fig.2).

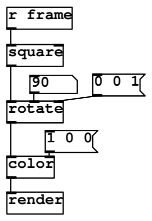

Figure 2: an early Gem-patch

A "source"-object (e.g. a `[square]`) generates a shape, which is then sent to one or several modifying objects (e.g. `[rotate]`) and finally the whole "signal" is sent to a `[render]`-object, which renders to the screen.

While this seems natural and simple, it is unfortunately not the way, how the underlying openGL-rendering engine works.

OpenGL is a state based system. This means that there is one single state that is manipulated by various modifiers (e.g. rotation). The modifiers do not "know" the objects they are modifying as they are only modifying an abstract state (represented by a transformation matrix). Only in one of the last stages of the display-process, the state is applied to the actual vertices that are defined by the data-source.

To achieve the illusion of a signal-flow environment, a *gemList* is passed between the objects. Each object adds its parameters to the *gemList*, e.g. the `[rotate]`-object would add `r` *`angle x y z`*. Finally the *gemList* is parsed and executed by the the `[render]`-object.

For a detailed explanation of this early model, see Danks (1996).

3 A hierarchical model (1997–2003)

The first attempt to integrate 3D-graphics did not mirror the underlying openGL-process, which led to the use of "display lists and other convoluted methods" (Danks (1997b)).

When Gem was ported from Max to pd, a complete redesign of the system changed the model from "bottom-up" to "top-down", which corresponds to openGL's state machine. The drawback of this change is, that nowadays Gem-patches are fundamentally different in structure from "normal" pd-patches, which makes them harder to understand for beginners.

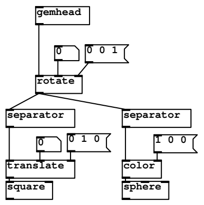

Figure 3: a hierarchical Gem-patch

Each rendering-chain now starts with a `[gemhead]`, which is ensures that all subsequent operations have a valid display-context. The `[gemhead]` is connected to several manipulation objects (like `[rotate]`) which directly change the openGL-state when called. Finally, a "vertex-emitting" object, like `[square]`, draws it's vertices within the current state (see fig.3).

3.1 Architecture

(Danks 1997a) The Gem-side core of the rendering engine is the *GemMan*ager. It holds the scheduler and the display-management, including the openGL rendering context and viewpoint settings.

To control the *GemMan*, the object `[gemwin]` is used. When there are several `[gemwin]`s, they all manipulate the same *GemMan*.

The `[gemhead]`-object is the beginning of a render-chain. On creation, it registers itself to the *GemMan*, so that it gets called each render-cycle. When the `[gemhead]`-object is destroyed, it unregisters itself from the *GemMan*.

When rendering is turned on, the *GemMan* will periodically call the `[gemhead]`s in the apropriate order to render their chain.

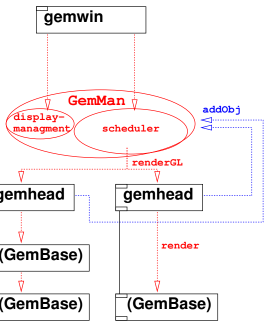

Figure 4: the core components of Gem

3.2 [gemhead] registration

There is no object representing the *GemMan*, rather it is created when Gem is loaded and stays statically in memory. Thus a `[gemhead]`-object can register itself to the *GemMan*, even if it is the first object created.

The registered `[gemhead]`s are internally stored in a linked list, sorted by their "priority"-value.

3.3 [gemwin] controls

Most parameters of the *GemMan* – like camera-position, fog,. . . – are static variables which can be set by the `[gemwin]`-object.

GemDAG The *GemDAG* (DAG="directed acyclic graph") is used to compile the network of gem-objects. Each gem-object registers itself to the DAG on receiving the `gem_state`-message when rendering is turned on. When rendering is turned off, this network will be destroyed. The compiled network is not aware of any changes made after compilation, therefore it is not possible to add new objects to or delete objects from the render-chain at runtime.

GemState The *GemState* is a a structure that holds a couple of variables that are passed from one gem-object to the object connected to it. These include general information on the rendering-mode we are in (whether lighting is enabled and whether we want smooth or flat shading). These general flags are re-set by the *GemMan* each render-cycle. Apart from that, the *GemState* includes complex data, that is passed between objects, namely a pointer to an *image* for pixel-processing and information needed for textures.

Finally, it holds a *dirty* flag that indicates that the gem-chain has been modified since the last render-cycle and thus needs to be rebuilt.

GemCache is a portion of memory that is shared between the objects of a render-chain. It is a means of telling the `[gemhead]`, that an object has changed the render-chain in such a away, that upstream objects are affected. This is particularly important for image-processing. As Gem tries to reduce the computational load, pix-objects are only executed when the upstream image changes. If the parameters of a pix-effect are changed, the image-source has thus to be told to resend the image.

The *GemCache* also manages the deletion of gem-objects from a compiled DAG. Whenever an object is deleted, the DAG becomes invalid to avoid segmentation faults.

3.4 Step by step

1. Gem is loaded; *GemMan* is created
2. a new `[gemhead]` registers itself to the *gemheadLink*-list in *GemMan*, which is ordered by the priority of the `[gemhead]`s.
3. a `[create(`-message is sent to `[gemwin]`: *GemMan* creates a new window and binds an openGL-context to it
4. Starting the rendering
   (a) When rendering is turned on (and the window is already created), the *startRendering*-function of each registered `[gemhead]` is called
   (b) The `[gemhead]` creates a new instance of *GemCache* and *GemDag*. A pd-message with references to these two instances is output through the objects outlet.
   (c) A gem-object (which inherits from *GemBase* that is connected to the `[gemhead]`, stores the reference to the *GemCache* locally.
   (d) The gem-object registers itself and its *render*- and *postrender*-callbacks to the *GemDag*.
   (e) The *startRendering()*-function of the gem-object is executed for initialization.

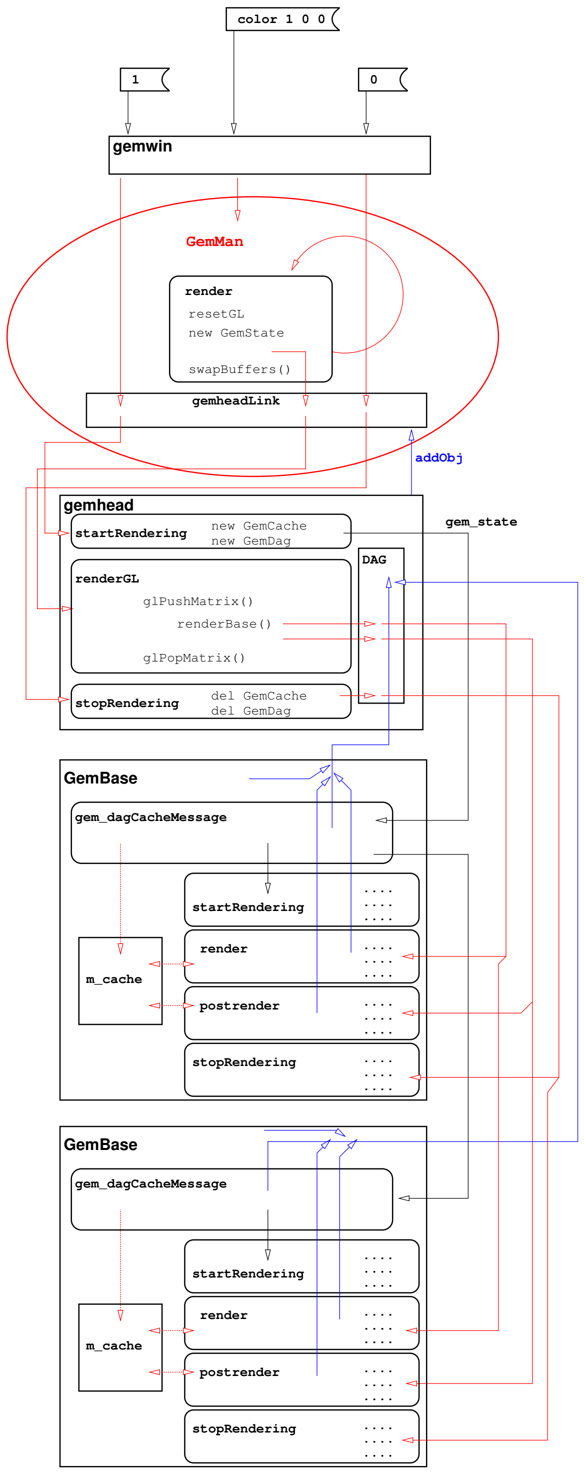

Figure 5: the render-process

(f) It then outputs a pd-message with references to the instances *GemCache* and *GemDag* to all connected gem-objects.

(g) With all gem-objects downstream the `[gemhead]` registering themselves to the DAG, a static branch of the render-network has been established.

(h) After all `[gemhead]`s have been called, the complete render-network has been created.

(i) If running in double-buffered mode, the *GemMan* calls its *render()*-function.

5. Rendering

   (a) Whenever *GemMan*'s *render()*-function is called, it first resets the openGL state machine (viewpoint, lighting, ...)

   (b) An instance of *GemState* is created and initialized (e.g. the lighting-flags)

   (c) The *renderGL()*-function of every `[gemhead]` is called with the *GemState*.

   (d) The `[gemhead]` checks whether the DAG and the Cache are valid or whether rendering has been disabled for this render-chain.

   (e) The current openGL-state is pushed to a stack.

   (f) The *dirty*-flag of the *GemState* is set to the value of the *GemCache*.

   (g) The *render(GemState*)*-Function of each entry in the DAG is called (top-down).

   (h) When the end of a DAG has been reached, the *postrender(GemState*)*-functions of its entries are called (bottom-up)

   (i) The original openGL-state is popped back from the stack.

   (j) After all `[gemhead]`s have been processed, the back- and front-buffers are swapped (if in double-buffered mode).

   (k) Finally, the next render-cycle is scheduled. (in double-buffered mode)

6. Stopping the Rendering

   (a) The re-scheduling of the *render()*-command is supressed

   (b) The *stopRendering()*-function of each registered `[gemhead]` is called.

   (c) `[gemhead]` calls the *stopRendering()*-function of each entry in its DAG, which sets the local *GemCache* of the gem-objects to invalid and allows to de-initialize.

   (d) The DAG and the *GemCache* are deleted.

4 Utilizing pd's message-system (2003–now)

(Zmölnig and Danks 2002)

The use of statically compiled DAGs has several big disadvantages. It is not possible to add objects once the DAG is compiled. If an object is deleted from the DAG, the whole DAG is set invalid, so that this render-chain is not rendered any more.

- The static rendering-network makes editing a Gem-patch very uncomfortable. Each time a modification is made to the render-chain, the rendering has to be restarted to see the results.
- The static nature also prohibits to change the render-graph dynamically. It is not possible, to decide at run-time which parts should be rendered.
- PD's messaging system is mirrored by the *GemDag*.
- It is not possible to use pd's objects for controlling the "signal-flow" (e.g.: `[spigot]`).

Therefore, the *GemDag* has been deprecated in favour of the pd-internal message-system. These changes only affect the `[gemhead]` and `[`*`GemBase`*`]` objects.

4.1 Starting and stopping the rendering

When rendering is started, a `[gemhead]` emits a message `gem_state 1`. This message triggers the execution of the *startRendering()*-function of a connected gem-object. After the gem-object has executed its render-initialization, it emmits the `gem_state 1` to tell all downstream-objects to execute their *startRendering()*-function. When rendering is turned off, `[gemhead]` emits a `gem_state 0` message. This triggers the *stopRendering()*-function of a connected gem-objects, which then outputs the same message to all connected objects.

4.2 Doing the rendering

5 Future renderings

(Zmölnig 2004)

6 Conclusion

References

Danks, M. (1996). The graphics environment for max. In *Proceedings of the International Computer Music Conference*,

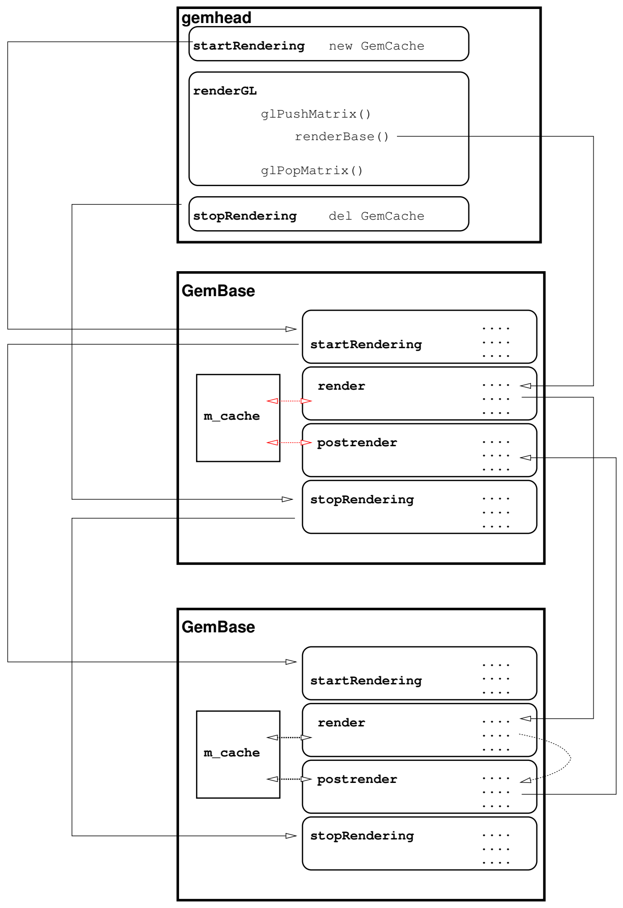

Figure 6: the render-process using pd-messages

pp. 67–70. International Computer Music Association.

Danks, M. (1997a). Gem-0.70 source code. *`ftp://ftp.iem.at/pd/Externals/GEM/BAK/OLD/gem-linux-0.70.src.tar.gz`, accessed 24.09.2004*.

Danks, M. (1997b). Real-time image and video processing in gem. In *Proceedings of the International Computer Music Conference*, pp. 220–223. International Computer Music Association.

Zmölnig, J. M. (2004). Gem-cvs source code, multiple_window-branch. *`cvs://cvs.gem.iem.at/cvsroot/pd-gem/Gem`, accessed 24.09.2004*.

Zmölnig, J. M. and M. Danks (2002). Gem-0.87 source code. *`ftp://ftp.iem.at/pd/Externals/GEM/gem-0.90.0.tgz`, accessed 24.09.2004*.