Gem for pd - recent progress

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

Abstract

Gem (Graphics Environment for Multimedia) is a library for Miller S. Puckette's graphical realtime computermusic environment "pure-data", that allows graphics processing based on openGL. While Gem has been available for several years, a lot of significant changes have been made in the recent past. The number of supported platforms has been expanded to apple's macOS-X. This paper also describes the fundamental changes that have been made to the architecture of Gem, allowing the use of dynamic render-graphs. Furthermore, a brief description of new functionalities in the realms of both image-processing and 3D-rendering will be given.

1 Introduction

Originally, Gem was written by Mark Danks as *Graphics Environment for Max* (Danks 1996). When Miller Puckette started his project *pure-data* (pd) in the mid-1990s (Puckette 1997), Mark Danks ported Gem to this new system, renaming it to *Graphics Environment for Multimedia*. In 2001, Mark Danks retired from active development of Gem. Since then, this package is maintained at the *iem*, Graz.

Gem does graphics processing within the framework of pd. To achieve this, it utilizes SGI's openGL, which is designed for 3D-graphics (Danks 1997). Since the late 1990s, enduser graphic cards with hardware acceleration for openGL have been available. This made it possible to do complex 3D-graphic rendering in realtime – e.g. with Gem – not only on dedicated graphic workstations but on any consumer machine. For performance reasons Gem had utilized static render-graphs, which could not be modified at render-time. With modern hardware this has become impractical, since the rendering engine had to be restarted every time, a new Gem-object was added to the graph Therefore Gem's graph builder has been rewritten to be able to use dynamic render-graphs.

Gem provides several basic geometric objects (like a *cube*), that can be used in 3D-space. While these objects are very simple to use, they cannot be modified arbitrarily (e.g: it is not possible to dislocate one single corner of the cube). Therefore a set of objects has been added, giving the adept user control to virtually any openGL-function.

With the advent of digital cameras and ever faster computers, it has become possible, to do high quality digital realtime image processing. The design of Gem's pixel processing has proven to be sub-optimal, since colored images are processed in RGBA-format, which is very consumptive in terms of memory and CPU-power. Gem now has support for a packed pixel format, which speeds up processing significantly.

2 Dynamic render-graphs

In prior versions of Gem, the render-graph was compiled into a set of static graphs when the rendering (e.g: graphics output) was turned on. This allowed the use of efficient *display-lists* (Neider et al. 1993). However, there are some major drawbacks to this strategy:

- Editing a patch while rendering is turned on is not possible. In the simplest case, the change is just not recognised by the system. E.g. after disconnecting an object from the patch-representation of the render-chain this 3D-object will still be displayed until the render-engine is restarted (and thus the render-graph is rebuilt). In the worst case (when objects are deleted) static render-graphs can lead to segmentation faults of the system.

- The display-list has to be recompiled by the openGL-system each time some property of the list is modified (e.g: rotation). This speeds up rendering significantly for static scenes, but imposes a slow-down upon rendering of dynamic scenes, where 3D-objects are moved continuosly.

- It is not possible to change the whole render-graph dynamically. While pd offers objects for controlling the data-flow (like the gate-like `[spigot]`), these do not affect the render-chain once compiled. It is thus not

possible to disable (at run-time) parts of the render-tree or to render subtrees multiple times without specially provided objects.

2.1 Controlling the signal-flow

To overcome the disadvantages of static render-graphs, Gem now uses pd's internal message-engine at run-time (as opposed to "compile-time" in prior versions). This basically means that rendering is done as follows:

1. each render-cycle, the `[gemhead]` objects receive a trigger command to render "their" subtree. On receiving the trigger, the `[gemhead]` sets up its internal state and outputs a *gemList*, a pd-message that holds meta-data for Gem's internal use.

2. a Gem-object gets the *gemList* by its first inlet. The *render()*-function of the object is executed.

3. the objects sends out the (eventually modified) *gemList* by its first outlet, stimulating connected Gem-objects to execute there *render()*-functions.

4. after all subtrees have been rendered, control is passed back to the Gem-object, which executes its *postRender()*-function.

This allows Gem to take advantage of all the pd-objects for controlling the message flow.

It is therefore possible to make a patch that holds a render-graph, a part of which is substituted during runtime (as shown in figure 1).

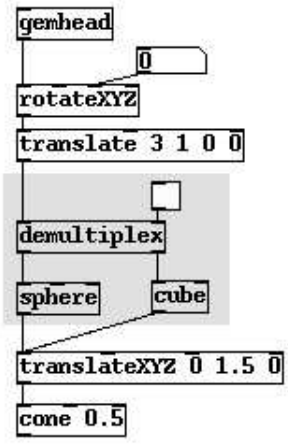

Figure 1: dynamic substitution of `[sphere]` with `[cube]` via a `[demultiplex]`-object

Furthermore, it finally allows programming Gem in real-time, e.g. the rendering engine doesn't need to be restarted everytime a new object is connected to the render-graph.

This procedure slows down the rendering process a bit, mainly because of an overhead due to pd's way of handling messages and calling object-functions. When doing image-manipulation, this overhead is quasi infinitesimal compared to the amount of pixel-calculation, and can therefore safely be neglected.

Things are a bit different when using solely openGL, as relatively few openGL-functions are called by a single Gem-object. The worst performance can be expected of the extreme case of a *1:1*-openGL-wrapper, as is shown below.

In most cases, however the difference will be imperceptible.

2.2 Particle systems

Another advantage of the direct usage of pd's message-system is, that it allows re-rendering of subtrees. The simplest way to do so is by connecting a Gem-object multiple times to a subtree (figure 2).

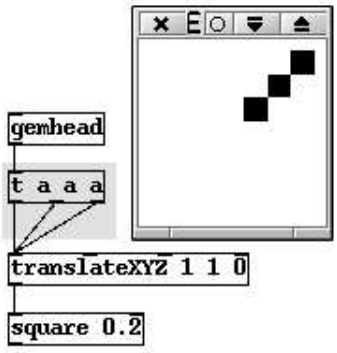

Figure 2: rendering a subtree multiple times

This method can also be utilized to create particle-systems. Particles can be considered as a group of similar 3D-objects with separate properties, like position or color. Gem has had a set of objects for manipulating particle-systems for several years, however, the actually drawn particles had to be either points or lines, which has proven rather unsatisfying.

With the new system, it is possible to render any possible render-graph as a particle (as shown in figure 3).

3 Pixel graphics

Because the drawing-engine of Gem is based on openGL, it is eminently suitable for creating 3D-scenes. Important for

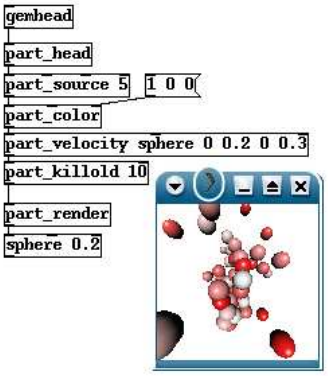

Figure 3: particle systems

3D-scenes is the ability to texture 2D-images ("pixel-graphics") onto a 3D-object.

Thus Gem has provided objects to load and texture images. Additionally it has been possible to apply some effects on this image-data, like inverting or mixing two images together.

With computers getting faster and video-capture-devices at hand, this set of objects has steadily been growing over the last years, allowing analysis and processing of still-images as well as video-streams, either pre-recorded or taken from a live-stream like a camera.

3.1 Colorspaces

Traditionally, Gem has supported two different colorspaces: `GL_LUMINANCE` for greyscale images and `GL_RGBA` for colored images.

The drawback of using RGBA is, that it consumes a lot of memory, because each colored pixel needs 4 bytes for storage. Thus another colorspace has been introduced in Gem to allow image processing with less memory (and less CPU) consumption: UYVY (also referred to as YUV 4:2:2), which stores information for two colored pixels into four bytes (therefore taking only half of the memory of RGBA). Since this is the only YUV-model that is supported by Gem, it is referred to as *the* YUV-colorspace.

While YUV is a good choice for colored images without a transparency channel, there is one major drawback: the openGL-standard does not support any YUV-colorspaces by default. Fortunately, on some systems the openGL-engine has been extended to support such images. Gem uses these extensions if available. If not, the image is transformed to a native openGL colorspace, just before the data is passed from main memory to the rendering engine.

3.2 Data-acquisition

Pixel-data can be obtained from various sources: simple images (like JPEG- or TIFF-images), pre-recorded movies (AVI, QuickTime or MPEG-encoded) and live-input. The live-input can be fed by TV-tuners, analog capture-devices or IEEE1394-cameras.

With the introduction of a second colorspace for colorized images, the question arises, how to acquire images in a certain colorspace.

Some colored images might be in RGBA-mode natively. Video-data (either pre-recorded or live) will most often be presented in some kind of YUV-format (but not necessarily UYVY)

Formerly, the answer was simple: if the image had color-components, RGBA was choosen, else the colorspace was Greyscale. Nowadays, the user can instruct a pixel-source object to present the data in the desired colorspace.

To change the colorspace of an image during processing, objects like `[pix_grey]` can be utilized.

3.3 Pixel-manipulation

Apart from the obvious differences like lack of color in greyscale images and no support for transparency in non-RGBA images, the user doesn't have to anything about colorspaces. Each object for pixel-manipulation, apart from some exceptions where providing methods for a certain colorspace makes no sense (like operations based on the alpha-channel), provides methods for all supported colorspaces. These can differ greatly in terms of CPU-consumption.

Optimizations (on the C++-level) can be done for each colorspace individually, or, if the effect-algorithm allows it, generically for all colorspaces. Since data in both the RGBA- and the YUV-colorspace can be memory aligned, SIMD optimization, such as MMX (Mittal, Peleg, and Weiser 1997), SSE2 or AltiVec (Fuller 1998), can be applied.

The set of objects for image-manipulation has become rather large (almost 100 object are dedicated to the pixel domain).

Most notable are those objects which can be used as mere "video-effects", like kaleidoscopic or motion-blur effects.

There is also a set of objects for storing large numbers of images in main RAM, which can then be accessed randomly. This, for instance, allows the construction of live video loops.

4 A wrapper for openGL

The rendering engine of Gem is entirely based on openGL. This fact, however, is normally transparent to the user. For instance, to display a geometric object like a square, it is enough

to attach a `[square]`-object to the render-graph. Then Gem will set up the rendering engine to draw a square, with the current color and transformation. Additionally, parameters for texture-mappings are set.

While hiding the complexity of openGL behind a simple interface makes it easy and fast to create simple scene descriptions, it gives away a lot of the power of the compex system that lies behind.

To give this power back to the adept users, Gem now offers a wrapper to each function of the openGL-library, up to version 1.2. openGL-functions can be accessed via objects that have the `GEM`-string prefixed to the function-name.

This approach is very similar to that of the DIPS environment (Matsuda and Rai 2000), which provides basically the functionality of Gem to jMax.

Each argument of the openGL-function can be set via creationarguments to the Gem-object and changed at runtime via separate inlets. For instance, the function `glVertex3f()` is wrapped into the Gem-Object `[GEMglVertex3f]`. Since this function takes 3 float-numbers as arguments, the corresponding object has three inlets for floats (in addition to the default inlet for the *gemList*).

An example of how to use the openGL-wrapper-objects is shown in figure 4.

Figure 4: using openGL-functions within Gem

The use of these objects is less than optimal in terms of performance. There appears to be a performance loss of about 50% (see figure 5). This is due to both pd's message system and the heavy use of indirect function calls of such an openGL-wrapper: calling an openGL-function from within C/C++ results in exactly one function-call that is passed to the rendering engine (which is very fast when hardware accelerated and thus neglectible in terms of CPU-consumption). Using a wrapper object, first the wrapper-function is called which in turn calls the openGL-function, resulting in basically two function-calls compared to a single one when writing compiled code.

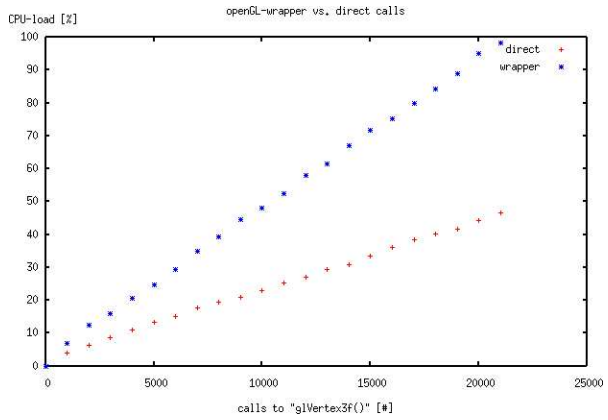

Figure 5: performance loss of wrapper objects

However, since Gem provides the framework for window-handling and image-acquisition, the availability of these wrapper objects withing the pd-environment allows for rapid prototyping of openGL-code, without the need to compile.

In production environments one would then probably choose to put the prototype into a compiled object.

5 Availability

Originally, Gem, like pd, was available only for SGI/Irix and Windows-NT systems. Günter Geiger has ported both packages to linux quite soon, which made them available on a wide range of hardware-platforms. After Apple released their new unix-based operating system macOS-X in 1999/2000, another port of pd was done by the community, which was essentially finished in 2001. With a running pd, James Tittle started to port Gem to the Macintosh-platform, which he finished in the year of 2002.

Since then, Gem is available on the three major operating systems that are in use nowadays:

- the Windows-family (ranging from Windows98 to WindowsXP)
- macOS-X ($\geq$ 10.2)
- linux

Currently the SGI-version of Gem not actively maintained, although it shouldn't be too hard to compile the package on an IRIX-system, since recent changes have not touched the system-specific implementations such as window-handling.

5.1 Paradigm: Crossplatform

One of the major strengths of Gem has always been it's cross-platform availability. This is mainly due to the fact, that the rendering engine is based on the system independent graphics library openGL[1].

Pd-patches are saved in a format, that can be read on all platforms. Thus Gem-patches (e.g: pd-patches containing Gem-objects) can be used on all supported platforms too, no matter on which operating system they were originally created. Operating-system specific details are made transparent to the user.

However, there are some minor platform-dependant differences of the various Gem-builts, especially where image-acquisition is involved. This is mainly due to the fact, that differenct decoding libraries are the default on different systems: e.g. on Windows the AVI-format is used for storing movies, while on Apple-computers there is a preference for QuickTime.

Gem on Windows : For decoding movie-files in AVI-format, Gem uses Microsoft's *Multimedia SDK* (Microsoft Corporation 2003a), which is supported by all Windows-platforms since Windows95. However, this framework is rather old and does not work with some modern codecs available. For decoding QuickTime-files, the *QuickTime For Windows*-library (Apple Computer Inc. 2003c) is used. Video grabbing now utilizes Microsoft's *DirectShow* (Microsoft Corporation 2003b), which enables Gem to use virtually all modern capture devices.

Gem on macOS-X : On macOS-X, image data of various sources can be acquired by using the standardized quicktime-interface (Apple Computer Inc. 2003b). This means that any image-source (e.g: live devices, like IEEE1394-camcorders or webcams, and movie-files with all sorts of codecs) that has Quicktime-support (Matsuda, Miyama, Ando, and Rai 2002) can be accessed from within Gem. Furthermore, there are macOS-X specific openGL extensions (Apple Computer Inc. 2003a), allowing highly optimised use of textures, such as direct use of YUV 4:2:2 images as textures: so image data can be acquired, processed and displayed without any colorspace conversion (Heckenberg 2003). Furthermore, most Gem-objects for image manipulation have already SIMD optimised processing kernels for both G4- and G5-machines, which speeds up image processing significantly.

Gem on linux : Under linux, things are a bit different, as there is no common interface for accessing various hardware devices and video codecs. Therefore a lot of different libraries are utilized. For capturing live image data, Gem uses both the *Video For Linux*-API (Dirks et al. 2003) – which allows the use of a large number of analogue capture-devices, such as TV-cards and webcams – and the *libDV*-API (Krasic and Walthinsen 2003) for access to digital video devices via Firewire. For decoding pre-recorded movies, Gem supports various libraries including the *avifile*-library (Langos, Kabelac, Yamashita, et al. 2003), *QuickTime For Linux* (Heroine Virtual Ltd. 2003b), *MPEG3* (Heroine Virtual Ltd. 2003a) and *FFMPEG* (Bellard, Niedermayer, Pulento, et al. 2003).

[1]There are several systems for description and rendering of 3D-scenes. The most prominent of these are Microsoft's direct3D (a part of directX) and SGI's openGL. One could argue, which one is better in terms of performance, flexibility and design. Basically both systems are supported by modern graphics-cards, which means, that the actual rendering is done by the GPU of the graphics-card and the CPU of the host-system can be used for other things. However, there is one big difference between the two systems: direct3D is a proprietary standard that is upported by Micorsoft's Windows only. openGL is an open standard that is available on practically every modern operating system.

6 Conclusions

The possibilities of Gem have been extended vastly in the last years. The increased set of objects for real-time video processing, makes Gem a suitable tool for VJs and media artists, who want to use pd's Max-like patch-system for simple chaining of video-effects. Users with good knowledge of openGL can build their own 3D-objects without the unavoidable limitations of pre-made objects and without the need for a compiler on a wide range of platforms. The usage of pd's message-system now allows truely dynamic render-scenes.

However, there are also a lot of things that could and should be improved. For instance, it should be made possible to render a scene to more than just one window. Gem doesn't yet utilize new functionalities in the openGL-standard, such as vertex-shading, a powerful technique supported by a number of new graphics card. Exchanging of data with other realtime video applications should be made possible, to benefit from their virtues, without having to build everything into a monolithic application. Under Windows, the usage of a modern pixel-acquisition API – such as DirectShow – has to be enforced, to allow the play-back of all modern video-file formats.

And finally, some bug fixing and a lot of optimisation has still to be done.

7 Acknowlegment

The original version of Gem has been written by Mark Danks. Since the maintainance of Gem has been passed to the the author of this article, the core development team has

grown: thanks to Chris Clepper (macOS-X), Daniel Heckenberg (windows), Günter Geiger (linux) and James Tittle (macOS-X) for their contributions. Of course it should be stated, that since Gem is an Open Source project, a lot of programming and bug-fixing has also been done by the pure-data community.

References

Apple Computer Inc. (2003a). Opengl extensions guide. In *http://developer.apple.com/opengl/extensions.html, accessed 09.03.2004*.

Apple Computer Inc. (2003b). Quicktime documentation. In *http://developer.apple.com/documentation/QuickTime/QuickTime.html, accessed 09.03.2004*.

Apple Computer Inc. (2003c). Quicktime for windows documentation. In *http://developer.apple.com/documentation/QuickTime/QuickTimeforWindows-%date.html, accessed 10.03.2004*.

Bellard, F., M. Niedermayer, J. J. S. Pulento, et al. (2003). Ffmpeg multimedia system. In *http://ffmpeg.sourceforge.net/, accessed 09.03.2004*.

Danks, M. (1996). The graphics environment for max. In *Proceedings of the International Computer Music Conference*, pp. 67–70. International Computer Music Association.

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

Dirks, B., G. Knorr, J. Schoeman, et al. (2003). video4linux hq. In *http://bytesex.org/v4l/, accessed 09.03.2004*.

Fuller, S. (1998). Motorola's AltiVec™technology. In *http://e-www.motorola.com/files/32bit/doc/fact_sheet/ALTIVECWP.pdf, accessed 10.03.2004*.

Heckenberg, D. (2003). Using mac os x for real-time image processing. In *Proceedings of the Apple University Consortium Conference*, pp. 125–134. Apple University Consortium.

Heroine Virtual Ltd. (2003a). Mpeg access for editing. In *http://heroinewarrior.sourceforge.net/libmpeg3.php3, accessed 09.03.2004*.

Heroine Virtual Ltd. (2003b). Quicktime for linux. In *http://heroinewarrior.com/quicktime.php3, accessed 09.03.2004*.

Krasic, C. and E. Walthinsen (2003). Quasar dv codec: lidbv. In *http://libdv.sourceforge.net/, accessed 09.03.2004*.

Langos, H., Z. Kabelac, H. Yamashita, et al. (2003). Homepage of the avifile-project. In *http://avifile.sourceforge.net/, accessed 09.03.2004*.

Matsuda, S., C. Miyama, D. Ando, and T. Rai (2002). Dips for linux and mac os x. In *Proceedings of the International Computer Music Conference*, pp. 317–320. International Computer Music Association.

Matsuda, S. and T. Rai (2000). Dips : the real-time digital image processing objects for max environment. In *Proceedings of the International Computer Music Conference*. International Computer Music Association.

Microsoft Corporation (2003a). Windows multimedia. In *http://msdn.microsoft.com/library/en-us/multimed/htm/_win32_windows_multimedia_start_page.asp, accessed 10.03.2004*.

Microsoft Corporation (2003b). Windows multimedia. In *http://msdn.microsoft.com/library/en-us/directx9_c/directX/htm/directshow.asp, accessed 10.03.2004*.

Mittal, M., A. Peleg, and U. Weiser (1997). MMX™technology archtitecture overview. *Intel Technology Journal, 3rd quater 1997 http://developer.intel.com/technology/itj/q31997/articles/art_2.html, accessed 10.03.2004*.

Neider, J., T. Davies, and M. Woo (1993). *OpenGL Programming Guide*. Addison Wesley, Reading, Mass.

Puckette, M. S. (1997). Pure data. In *Proceedings of the International Computer Music Conference*, pp. 224–227. International Computer Music Association.