Gluing Things Together - Scheme in the Real-time CG Content Production

Shiro Kawai*
Square USA Inc.

Abstract

Cutting-edge 3D real-time rendering is where programmers rack their brains to feed more instructions to CPU or transfer more data on the bus within each frame, and dynamic languages seem to have no place there.

However, there is another hidden cost which is growing rapidly as the quality and complexity of rendered images get higher—the cost of content creation.

At SIGGRAPH 2000 and 2001, we, Square USA R&D team showed real-time rendering of scenes from full computer-generated movie “Final Fantasy: The Spirits Within”. Although we used the most advanced rendering hardware at that time, such as Sony Computer Entertainment’s GSCube parallel rendering engine and nVidia’s Quadro DCC chip, we had to spend lots of time to tune data to achieve the maximum image quality.

We found it tremendous help to have an embedded Scheme interpreter in real-time rendering engine. Scheme could lay out the data on the memory, inspect the state of the engine, and handle user interactions and allow data manipulation over the network, without affecting rendering performance. Scheme representation of the scene structure worked nicely for the last-minute tuning (lots of Emacs works, in fact).

1 Introduction

It is not a news for Lisp programmers that Lisp is used in the interactive computer graphics. Actually, Lisp used to be *the* language to do graphics. There were various reasons for it, but we think one of the reasons was that dealing with the data structure in computer graphics is basically tweaking a directed graph dynamically, and Lisp was very good to deal with graph structure at run time.

However, real-time interactive computer graphics are constantly facing the limit of hardware due to ever increasing requirement of pushing more data to the graphics hardware, and such languages that the programmers can have precise control over what’s on the memory and in the instruction stream are much preferred—so C and C++ have dominated the field. The dynamic languages tend to bring unpredictability into the tightly-tuned engine and are frowned away by many programmers. (Actually, some game programmers even avoided C++ until recently, because it is prone to introduce unexpected pointer indirection and hidden fields in the data structure).

Nevertheless, tweaking the parameters and structures in the rendered scene at run time is inevitable during the production stage. People deal with it starting from “run-time configuration file” that sets some parameters at the startup. It is natural that such configuration file syntax evolves to a simple command language, then to a full-fledged programming language. Square Co., the parent company of Square USA, has been using in-house specialized scripting languages to describe game events for long time. Recently there are some attempts to embed full-feature language, such as Python[1], or Lisp[2], in the game engine.

Square USA R&D Team had been developing an in-house real-time rendering engine in a project called Dancer, and we adopted Scheme as an embedded scripting language. The engine and scripting features were tested *in production* for “Final Fantasy in Real Time” demonstration in SIGGRAPH 2000 and 2001 exhibition. The demonstration showed real-time rendering of a sequence taken from a full-CG Movie in photo-realistic quality, and that made us face some nontrivial challenges[3].

This paper describes our experience in the production of those demonstrations, and how we addressed the issues using Scheme as a glue language.

In the next section we’ll give an overview of the Dancer Project in Square USA. In the following sections we describe the implementation details of the embedded Scheme engine, the problems we faced during the production of the demonstration, and how we used Scheme to address those issues. Then we summarize our experience, and conclude the paper by the future direction of real-time scripting.

2 The Dancer Project

The Dancer Project is an in-house development project in Square USA, aiming at building a high-quality real-time 3D rendering and animation engine that took advantage of advanced graphical hardware as much as possible. It started in 1999, when Sony Computer Entertainment (SCE) revealed PlayStation 2 (PS2) specification. The engine first targeted at PS2, as well as standard OpenGL, then the support for DirectX and nVidia’s OpenGL extensions were added.

*Now at Scheme Arts, L.L.C. `shiro@schemearts.com`

It was used for technical demonstration titled “Final Fantasy in Real Time” at SIGGRAPH 2000 and 2001 exhibitions, showing a scene from the movie “Final Fantasy: The Spirits Within” rendered in real time, on SCE’s GSCube parallel rendering engine (in 2000) and nVidia’s Quadro DCC graphics card (in 2001). Dancer was also used for a preview tool to support the production of “Final Fantasy: The Spirits Within” and another short animation. A part of the Dancer library was used in a console game title as well. (But the latter applications didn’t use the Scheme engine.)

The Dancer engine was implemented mostly in C, with some assembly language code. Performance and predictable memory consumption at run time were the highest priority. The use of C++ was rejected by the potential users of the library since it didn’t fit in their production pipeline.

One of the characteristics of the Dancer engine was its modular architecture. It consisted of mostly independent, modular libraries, and the user could take the features they needed for their application. Dancer had its own data structure, but it didn’t force the user’s code to use the Dancer structures which might not fit in the internal structure of the application. Instead, the Dancer API could take minimal structures, such as a pointer to an array of floats.

For example, the deformer module took pointers to the source and destination float arrays that represented vertex positions, and updated the destination float array every frame according to the settings. The hierarchical transformation module took pointers to the 4x4 matrices and updated them according to the transformation hierarchy. The rendering module used the shape structure that took a pointer to the vertex arrays which were an array of floats. They could be used together to create a complete animation engine, but each modules could be plugged-in to user’s application by just passing pointers.

3 The Scripting Engine

The Dancer’s scripting engine had to fit this framework. It had to work as a pure “add-on” module, that is, adding extra data field (such as tag bits) to the Dancer data structure or interfering with application’s memory management were not the option.

We started with STk[1] Scheme interpreter, for embedding it to other applications was relatively easy, and we had been using it for various in-house tools, including embedded Scheme plug-in for Maya[2].

First, due to the limitation of memory space (especially of PlayStation 2), we stripped unnecessary features:

- Only fixnums and flonums were used for numbers (no bignums).
- Flonums used floats instead of doubles.
- Some error checks were omitted.
- Continuations worked upward-only (just setjmp and longjmp).
- Dropped support of module system, promise, eval-hook, autoload, virtual ports, FFI, POSIX functions, dynamic loading, and unix processes.

PlayStation 2 and GSCube runtime had limited support of system functions, so some of them were omitted and some of them had to be emulated (e.g. we built a buffered file I/O on top of PlayStation 2’s file transfer protocol via host machine).

Memory management was tricky. Dancer data structure might point in the middle of other data structure, as, for example, a hierarchical transform node pointed to a matrix inside another object. Lacking precise type information and layout information of each object, there was no way for GC to know which objects were alive.

However, as the real-time rendering engine, the allocation pattern of Dancer data was usually very specific. Almost all data was laid out on memory during initialization stage, and the allocation during real-time playback was generally avoided. So we decided to make the Scheme engine not to reclaim Dancer data structure.

The scripting engine used its own memory allocator to allocate Scheme object from the designated memory space. The objects from the space was managed by GC. The Dancer objects were allocated by application’s memory allocator. The Dancer objects allocator could be called from Scheme to obtain the pointer of new Dancer objects as a foreign pointer. Scheme could also call the explicit deallocator of Dancer objects, although it was the responsibility of the script programmer to make sure the Dancer scene graph would be consistent after deallocation of some objects. The explicit deallocation was hardly used, since it was always easy to flush entire memory and reload the data into a fresh state.

The Scheme binding to Dancer library API was generated semi-automatically from hand-written API description format. The Scheme binding was fairly low-level, in the sense that it was allowed to deal with C pointers within Scheme. Scheme code could take the address of a field of Dancer object and put it to a field of another Dancer object, for example. It was accepted well by the programmers, since they could treat Dancer structures in the same way as in C.

The Scheme engine also had a couple of important features, one was UI binding and the other was a networked listener. The UI binding allowed Scheme code to construct menus, to connect keyboard and mouse events to Dancer objects, and to register callbacks to such events. Connecting events to Dancer objects means to pass the UI driver the pointer of a field of some Dancer object and ask the driver to update the field whenever the event occurs. The important fact here was that no Scheme code would run when actual events occurred, thus no allocation and no GC would be required during real-time playback. Callback feature could be used if we needed more than such a simple connection, mainly to implement the features useful during authoring stage, when a pause of GC was a much less problem. (In fact, it turned out that the pause of GC was never be a problem; we hardly noticed it. The pause caused by texture swapping was much more problematic).

The networked listener was a read-eval-print loop that lis-

[1] `http://kaolin.unice.fr/stk/`

[2] `http://www.aliaswavefront.com/en/products/maya/`

tened network connection. It used sockets on PC/nVidia platform, and a special protocol on PlayStation and GSCube platform. Initially we implemented it for the last resort of run-time tweaking, when we needed to adjust some parameters without stopping the playback. However, it turned out much more useful when we connected the Dancer playback engine and the authoring software. We'll explain it later.

4 The Production

We experienced two production process using Dancer and the scripting engine. The first one was for the exhibition of SIGGRAPH 2000, where we played back a sequence (called "BOA") taken from the "Final Fantasy" movie which ran about a minute and included a single character. It ran on SCE's GSCube parallel rendering engine, which essentially had 16 PlayStation 2's and an image composition hardware in it. The second one was for the exhibition of SIGGRAPH 2001, where we played back a sequence called "SLA", which ran about four minutes and had two characters. In both demonstrations, the audience could control the camera and the lights interactively while playing back the animation. In the 2001 demonstration the audience also could adjust various parameters, such as object colors and transparencies, interactively.

We had the movie data, but they were extremely large and complicated and there was no way to render them in real-time as they were. Thus the production was mainly to tune and reconstruct the data into a suitable form for real-time playback. It was not just a matter of reducing the amount of the data—sometimes the fundamental structure needed to be changed, such as animation setup, and sometimes the movie data was totally irrelevant and needed to be redesigned, such as lighting setup. The redesigning involved artistic decisions, and we worked with several artists to achieve the goal. The actual process was described in detail in [3].

There were some issues we wanted to address by scripting.

Integration of the data: The scene consisted of large number of data from various sources. For example, a character model was exported from Maya, and the texture was retouched by Photoshop. Animation data was also exported from Maya but it was chopped for each shot and each individual channels, and had to be concatenated somehow. The lighting data needed to be recreated in terms of the Dancer's rendering engine. We needed some way to integrate them, so that they form a scene graph in the Dancer's memory.

Shortening turn-around time: Traditional workflow for real-time content production is to iterate (1) exporting data from authoring tool (Maya), (2) loading the data into the target environment, and (3) checking the appearance and fixing the original data. It does not work well as the amount of data grows, for the exporting takes more and more time. In our production, the extreme case was the facial animation of the main character, which took 8 hours to export[3]. It was crucial to have the means of tweaking the data and seeing its effect immediately on the target environment, instead of repeating the above time-consuming iteration.

[3]This was because what we needed was not the source of the animation, which was several tens of animated real numbers, but the preprocessed intermediate data, which was animating several thousands of vertexes.

Supporting trial-and-error: When you want to achieve certain visual appearance, it is not uncommon that there are two approaches, one that puts more work on designer side to prepare data, or another that emphasize more on programmer's work to deal with the data. The cost and effect of these approaches may vary greatly, and trial-and-error is inevitable in the production. Nevertheless, the tight schedule of the production usually doesn't allow enough room for trials. It would have been a great help if we had the means to try out certain programmatic effects quickly, instead of waiting a day or two for a programmer to write up the complete code to do it.

5 The Roles of Scheme

We used Scheme in various places throughout the production, but it could be summarized as the following several topics.

5.1 Scheme as a file format

To address data integration problem, we just decided to use Scheme program as the integrated file format. So loading data files was just evaluating Scheme scripts.

We had a special binary format for individual data files such as geometry data and animation data. Scheme had API binding to read those binary files. After placing those binary data in the memory, the Scheme script was responsible to interconnect pointers of the data structures to construct the scene graph.

We wrote an exporter plugin in Maya that emitted binary files and a Scheme program to load and interconnect them. Not all data in the original scene were exported as binary; for example, shader and material information was exported as a Scheme program that would construct a shader node and set up the parameters.

Having a Scheme program as a file format gave us lots of flexibility. Splitting the data hierarchy into manageable size was trivial, for we could just 'load' the subtree of the data. Treating animation data was a bit more involved, for the original data was prepared for each shot, having the time frame beginning of the shot. The script had to load the animation data of shots in the sequence, then offset them accordingly within the sequence. Furthermore, simple concatenation of the animation data wouldn't work, since the interpolation around a shot boundary would yield wrong result. So we used a special Dancer structure called a "switch connector", which took input from animation channels of all shots, and one "control" input that selects the active shot, then output the selected data into the given pointer. A hand-written Scheme script took care of creating required switch connectors and wiring them.

One example of such flexibility was the ability of parameterization. It was often required to play back just a certain shot, or even a part of the shot, in order to tune some data

```
(define *shot-lengths* '((3  84) (4  402) (4a 48) (5  36) (6  74)
                         (7  113) (8  72) (9  120) (10 217) (12 120)
                         (13 368) (15 86) (16 128) (18 275) (18a 72)
                         ...))

(define *shot-data*
  (let ((frame 1))
    (map (lambda (len)
           (let ((result (list (car len) frame (+ frame (fps-frame len) -1))))
             (set! frame (+ frame (fps-frame len)))
             result))
         *shot-lengths*)))

(define (get-motions shots file-format-string)
  (define (read-channel file)
    (sqmo-channel-read (sq-complete-filename file)))
  (map (lambda (shot-data)
         (if (memq shot-data shots)
             (create-motion-channel
              (read-channel (format #f file-format-string
                                    (shot-number shot-data)))
              (first-frame shot-data))
             #f))
       *shot-data*))
```

Figure 1. Excerpt from the integration script, showing the part that defines shots in the sequence and then loads required animation channels ("motions"). The variable `*shot-lengths*` keeps assoc-list of shot name and its length. Changing the length of the shots or omitting certain shots were as easy as just changing the value of `*shot-lengths*`, which helped us greatly during data tuning and debugging stage.

or debug some defects. We had a single location that defines shots and their lengths (see figure 1), and changing the shot to play back was just a matter of changing the value of that definition.

It was also important that we had an ability to patch things procedurally; if we found some bug in the rendering engine that had a problem with certain data, we could add a Scheme function to the data file that scanned the data hierarchy and rewrote the problematic data, so that the artists could keep working without waiting the bug to be fixed.

5.2 Scheme as a debugging tool

Debugging was especially a problem in GSCube project. The target program ran on 16 CPUs in GSCube, while you could only login to the host processor that controls those CPUs. A remote debugger was provided but difficult to use, and we ended up mostly using print statement to monitor the state of CPUs.

Scripts came handy here. It didn't require recompilation, of course, but the real power came from the ability of registering callbacks to events, including shot boundaries or frame boundaries. So we could write a procedure, for example, that were evaluated in each frame while frame count was in a particular range and reported if the data was in supposed range or not. Evaluating callbacks required some memory allocation in the Scheme engine and might cause GC during playback, so it could have a probe effect, but in practice we didn't have the problem. Besides, we could also set a callback to some button event to turn on and off the debug stub, so that we could evaluate the probe effect itself.

Even in development on PC for SIGGRAPH 2001, where we could invoke debugger as we needed, it was sometimes useful to use a script to show certain values on the screen, for example, so that we could monitor the data during playback.

We expected the network listener to be another powerful debugging tool, but it wasn't used very much for debugging. It was probably because "inspect" feature of the listener was poor, and it took lots of typing to reach the data you want to investigate. We realized the power of interactive top-level was not just from the fact that you could evaluate the expression interactively, but also from the rich supporting tools such as inspectors, apropos, and history features. However, the network listener turned out to be useful to construct authoring environment, which we describe later.

5.3 Scheme for rapid prototyping

The demonstration UI was a kind of last-minute hack, since we were not sure how much we could show until very late in the production. The UI was written in Scheme, which helped a lot to tweak it until the last moment.

There were another incident that Scheme served as a handy tool. Lighting of the scene had to be redone for the demonstration, since the movie's lighting setting was not for real-time rendering. The adjustment must have been done *in* the Dancer's rendering engine; the authoring tool (Maya) couldn't reproduce the exact image that Dancer produced. So we quickly hacked a lighting tool on Dancer, which enabled an artist to change positions and parameters of several lights while the scene was playing back. The tool could also save and load the light settings. The tool features were entirely written in Scheme, without modifying Dancer engine code.

An artist worked on the tool and finished lighting of 20-some shots within a couple of days. Considering that it had taken many *weeks* to do lighting in the movie scenes, we realized the power of real-time playback for authoring environment. (This is not a fair comparison, though, since the movie scenes involved a lot more light settings than just 6 to 8 hardware-rendered lights in the demo. Still, the artist mentioned that it was much easier to see the effects immediately, instead of waiting an hour for the image to be rendered).

5.4 Scheme for authoring

The experience of lighting tool motivated us to proceed that direction further. Although it wasn't finished in time in the production of the demonstrations, we later developed a prototype system called "Laika", which was an attempt to make a fusion of authoring and playback environment.

The Laika authoring prototype consisted of an authoring tool (Maya), and the Dancer playback engine (see figure 2). Maya had our in-house embedded Scheme plugin (called "schemaya"), which monitored changes in the Maya scene and propagated them to Dancer, via Dancer's network listener.

Schemaya was also based on STk and could use STk's object system, STklos, which had CLOS-like MOP[4]. We wrote a simple distributed object system on top of it. An object in Maya side was defined as a "proxy", which had slots marked 'remote' (see figure 3). Changes of the remote slots were sent to Dancer side, and reflected promptly on the scene being played back. Proxy metaclass managed caching and optimizing the communication. The actual messages sent to Dancer were simply a bunch of `set!` expressions, which were evaluated by Dancer's network listener.

An artist could change camera setting and lighting in Maya, which would propagate to the Dancer's rendering immediately. Besides, Laika could use Dancer engine to render multiple images while jittering camera and lights, then compose them to create higher-quality images with nicer effects like soft shadows and depth-of-field.

Unfortunately, Square USA shut down the studio before "Laika" was used in the real production. However, we think this technique should be used in the future production.

6 Experience

Thinking back our experience of two demonstration productions, we'd like to note several points.

S-expression was a win. It could be any structured format with well-defined external representation, but I think S-expression is the simplest form among other possibilities. Reading and writing S-expression are trivial[4], and we wrote lots of small, throw-away Scheme scripts to fix the data deficiencies, since re-exporting the data from Maya was too expensive. Emacs has built-in support to deal with S-expressions, and the capability of hand-editing data files was indispensable for the last-minute crunch time.

Higher-order function was a win. This was a surprise for us. We expected scripting engine needed just trivial stuff, such as loops, conditionals, and function definitions at most. However, the tweaking of data produced lots of similar code, and some kind of parameterization was handy for refactoring them. And the functional abstraction was, again, the simplest way to realize that. For example, figure 4 shows an excerpt of the scene-setup code that creates a "switch connector" node which feeds different animations for each shot from different motion files. There were some variations of in types of animated data (matrices, colors, etc.) and the fields, so the connector creation function took two functions, `connector-builder` and `slot-accessor`, to parameterize the construction.

GC was a problem. The speed of GC was not a big problem, since no memory allocation was done while the tight rendering loop was running, so GC would never run in the normal playback. (The menu UI and remote evaluation could cause GC, but it was an acceptable pause in the production period. It was possible to set up the scene so that no allocation would be done during the actual demo.)

The real problem was that existence of GC might force a certain style of programming. If we had wanted to GC not only the Scheme data but also the actual Dancer structures, we would have had to ask the application that uses Dancer to obey some restrictions, such as using Dancer's memory allocator or limiting the use of pointers so that they would be visible from GC. This would severely limit the usefulness of Dancer, for the memory management is one of the core part the potential application writers would want to tune by themselves.

One possible way to address this might be to provide several choices of allocators that works with GC. For example, sometimes real-time rendering engine uses simple first-in-last-out allocator, where the memory is divided by free area and used area by a single pointer, and a new object is allocated by just taking the required chunk of memory from the top (or bottom) of the free area and updating the dividing pointer. The objects must be freed in the reverse order of allocation, or freed all at once[5], but such limitations are accepted for the better performance. It would be nice if we have GC that is aware of this type of allocator.

Limited memory space was a problem. We couldn't keep meta information such as the layout of Dancer structures and a catalog of instantiated objects in the target's memory, due to the limitation of memory space. It reduced the usefulness of interactive listener—we needed to chase pointers by hand, looking at the source code, to reach the object we needed to inspect, which was too cumbersome.

[4]It may be argued that the ease of parsing and generation is the matter of library support; say, if we have XML parser and generator, it would be just as easy as to deal with S-expr. Still I think S-expression reader can be written much smaller than XML parser, which is crucial if we embed it in the game engine.

[5]Freeing a chunk of data at once, in the end of a frame or a scene, is often the case.

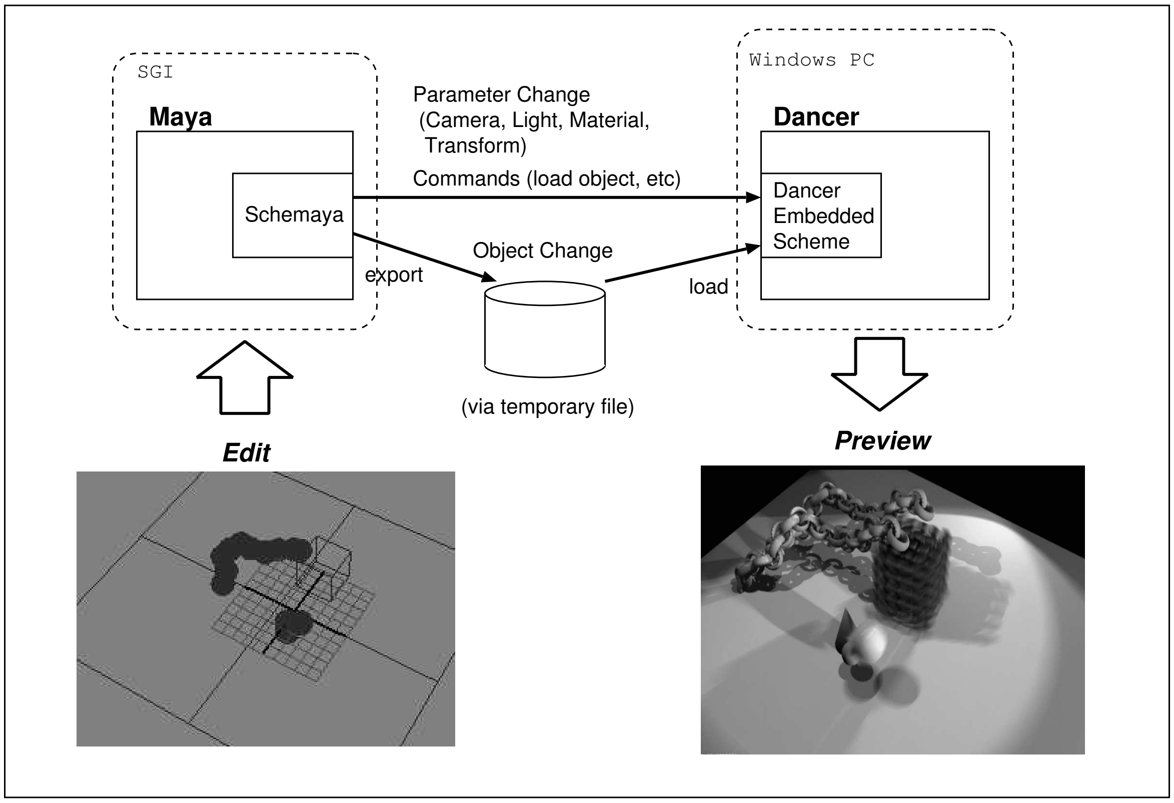

Figure 2. The "Laika" authoring prototype.

```
(define-class <dancer-light-proxy> (<proxy>)
  ((id :init-keyword :id :accessor id-of)
   (name    :accessor name-of    :initform #f :init-keyword :name)
   (state   :accessor state-of   :allocation :remote)
   (color   :accessor color-of   :allocation :remote)
   (target  :accessor target-of   :allocation :remote)
   (rotate  :accessor rotate-of   :allocation :remote)
   (distance  :accessor distance-of :allocation :remote)
   (fov     :accessor fov-of   :allocation :remote)
   (jitter-angle :accessor jitter-angle-of :allocation :remote)
   (jitter-distance :accessor jitter-distance-of :allocation :remote)
   (phong   :accessor phong-of :allocation :remote)
   (settings :accessor settings-of :allocation :virtual
             :slot-ref  (make-light-settings-getter)
             :slot-set! (lambda (o v) #f))
   )
  :getter-builder make-light-proxy-getter
  :setter-builder make-light-proxy-setter)
```

Figure 3. Definition of "remote" object. Changes to the slots marked remote would propagate to the other side and reflected in the scene being played back.

```
(define (create-switch-connector obj shots file-format-string
                                 connector-builder slot-accessor)
  (let* ((swc (connector-builder (list (slot-accessor obj))
                                 (if (string? file-format-string)
                                   (get-motions shots file-format-string)
                                   file-format-string)
                                 *studio*))
         (inc (sqst-create-integer-connector
                (sqco-switch-connector-get-current-source-ptr swc)
                *shot-step-motion*
                *studio*)))
    (sqst-studio-add-updater-before *studio* *root* swc)
    (sqst-studio-add-updater-before *studio* swc inc)
    swc
    ))
```

Figure 4. Using higher-order function for abstraction. The function `create-switch-connector` **takes two procedures,** `connector-builder` **and** `slot-accessor`**, that have information about the type and location of the data to deal with.**

On PC platforms this restriction could be removed, but the special platforms such as a console game engine will always have less memory than the PC at the same generation.

In the applications like Laika, it might be possible to keep the meta-information in the host machine. We could make the target machine report addresses of newly allocated object to the host machine (or even make host machine to emulate the allocator of the target machine), so that the host machine can keep a table of names and pointers of allocated instances, instead of letting the target machine do so. A special client program running on the host machine can then use the information to help interactive listener, much like remote debuggers do.

Power of abstraction must be advertised more. When talking about the ideas of embedded Scheme, it is common to get a reaction such as "Why do we need that complicated stuff, such as higher-order functions? Can't we keep it simple, just using variables and assignment command, for example?" The problem of this idea is that eventually some kind of abstraction would be required, and adding them in ad-hoc language would be horrible. In fact, Scheme may be one of the simplest form among the systems with the given abstraction capability.

7 Future

As the graphics hardware gets faster and the audiences' expectation to the real-time interactive graphics gets higher, the amount and complexity of data the rendering engine has to deal with will ever increase. It will be more important to push the point of fine-tuning toward the target platform, otherwise turn-around-time will be increased to the point being prohibitive for production to do enough iterations.

The cost of having full-featured scripting language inside the real-time rendering engine becomes smaller nowadays. We expect it to be the mainstream. With clever GC and supporting tools, Scheme can be a good choice for it.

We also believe the fusion of authoring and playback environment, not only in the consumer game production but also in the high-end computer graphics. Such applications will be required to deal with complex structure, spread out among processors and changing dynamically at run time. Lisp is still a good language for it.

Acknowledgments

The author would like to thank to Square Co., in Tokyo, Japan, to give the permission to present this material in the International Lisp Conference.

The author would also like to thank to ex-Square USA members, who participated in the ambitious project to create photo-realistic computer generated human animation. The company was gone, but what we learned will live and spread, pushing up the limit of this field.

8 References

[1] Bruce Dawson. *Game Scripting in Python*, in Proceedings of Game Developers Conference 2002, March 2002.

[2] Stephen White. *Postmortem: Naughty Dog's Jak and Daxter: Precursor Legacy*, in Gamasutra July 2002, `http://www.gamasutra.com/features/20020710/white_02.htm`.

[3] Kaveh Kardan. *Running A "Final Fantasy" Movie Sequence in Real Time*, in Proceedings of Game Developers Conference 2002, March 2002.

[4] E. Gallesio. *Stklos: A scheme object oriented system dealing with the tk toolkit*, In Proceedings of ICS Xhibition'94, San Jose, CA, pages 63–71, 1994.