Final report, Erl-King project.

Isaac Dimitrovsky, April 1 2004

Introduction

The following summarizes my experiences producing the updated Erl-King version that went on exhibit at the Guggenheim NY on 3/18/2004. To understand some of the material here you may need to refer to my original proposal in Appendix 1, and to the message from Jeff Rothenberg in Appendix 2. I've tried to be conversational in tone, and to include parts of the experience that would be useful to someone tackling a similar project in the future.

The Erl-King is a pioneering interactive video artwork by Grahame Weinbren and Roberta Friedman. It was originally created in the early 1980's using a Sony SMC-70 CP/M computer, three laser disk players, and assorted custom electronics and other hardware (see my proposal for a diagram of the original hardware organization). In October 2003 I was consulted about the feasibility of producing an updated version of the Erl-King that would allow it to be run in the future, since it was becoming more difficult over time to keep the original system in working order. The updated version was to be included in an exhibit in mid-February 2004 (this date was later postponed to March 18 by the Guggenheim staff). The Erl-King consists of an authoring system used to edit the video presentation, and a runtime system that presents it to the viewer.

The original vision for the Erl-King update, as expressed in the message from Jeff, was to use object-code level emulation to run the original SMC-70 operating system and the Erl-King programs. While this was an interesting idea in principle, I quickly came to the conclusion that it probably wouldn't be feasible in the time available:

- The CP/M operating system itself was quite popular and retains a cult following to this day, so quite a few CP/M emulators were available. However, each particular CP/M computer has system-specific peculiarities that must be tracked down to produce an emulator, and no ready-made emulator was available for the SMC-70.
- The Erl-King piece (and likely any other major CP/M based video work) required a lot of external hardware, so a working SMC-70 emulator would be half or less of the battle.
- The prospect of trying to debug the combination of a new CP/M emulator at the object code level together with assorted hardware emulators seemed very hairy.

When I looked at the original Erl-King it became apparent that the system was written relatively cleanly in a CP/M version of the higher-level programming language Pascal called MT+ Pascal. I therefore suggested a compromise solution that would emulate the external hardware devices, but interpret the original Erl-King software at the Pascal code level instead of emulating the object code. This proposal (see Appendix 1) was accepted on Nov. 10 after some debate, and I was able to get to work on the update.

First stage: Pascal interpreter and authoring system.

As stated in my proposal, I first wanted to find or write a Pascal interpreter and use it to get the Erl-King authoring system running. I thought this would be a good sanity check for the general plan, and could be done without having to deal with the video server programming or wait for video to be captured. After a day or two of research done mostly on the web, I decided to write the interpreter myself:

- I hadn't used Pascal in many years, but as I read a couple of the manuals it became clear that several essential features were left out of the original language specification (ex. random access files, pointer arithmetic, string operations). The result of this was that each particular version of Pascal such as the MT+ version used by the Erl-King had its own implementation of these features. Therefore, the open-source Pascal interpreters that I found were likely to require substantial modification before they were usable for the updated Erl-King.
- I felt it would be useful to have the full ability to modify the interpreter while I was debugging the system.
- I thought it would be easier to maintain the updated Erl-King if it was self-contained and didn't depend on external packages that would also be maintained over time.

This was a close call – if I had found a good open source Pascal interpreter for MT+ Pascal in particular I might have used it. In retrospect, this decision worked out well. I was able at several stages to customize the interpreter's behavior in useful ways that would probably have been difficult if using an existing interpreter. The following points refer to Pascal in particular, but similar considerations would probably apply to future projects, both interpreted and emulated:

- According to the rules of Pascal, certain bad behavior results in undefined results (ex. using variables that haven't been initialized, trying to use null pointers). In practice many Pascal programs unknowingly behave badly, and subtly depend on the particular results that this bad behavior has in the version of Pascal that they use. In these cases, the interpreter or emulator must in effect match the "undefined" behavior of the original system to give the same results. Several cases of this type were detected while implementing the Erl-King update.
- A related benefit of rolling your own interpreter or emulator is the possibility of doing some types of automatic error correction that were too inefficient for the original environment or for a general purpose interpreter. For example, in the Erl-King update I was able to include automatic garbage collection of memory, which in most cases will avoid running out of memory if the original Pascal source code has a memory leak.
- Most strangely, at one point I found a part of the Pascal source code that has what is clearly an error in syntax according to the Pascal grammar. For some reason, it was passed without complaint by the MT+ compiler on the SMC-70. After scratching my head a bit, I decided to be scrupulous about not modifying the source code and simply set my interpreter to accept the error as well.

The second point above is one case of an interesting question that came up repeatedly – where do we draw the line at modification of the original system's behavior? In the case above, I think the answer is fairly clear – if the original system has errors that would sometimes cause it to crash or otherwise clearly cause errors, this would probably not be considered an essential part of the original system experience that must be emulated! In other cases (ex. response time or image quality), the value of "improvements" to the original may be debatable. For example, the initial working version of the updated Erl-King had a substantially faster response time than the original to switches in the played video. The artist decided that this in fact degraded the viewer's experience, so we slowed down the video player switch to match the original speed.

One thing that surprised me at this point was the amount of useful information I was able to get off the web. As I mentioned above, I found that CP/M retains a loyal following to this day, and so there were a number of more or less fascinating websites containing history, manuals, and other information. Among other things I was able to find a Pascal MT+ manual with much more detail than the one from the original system, which was a great help in writing the interpreter.

The next question was what language to use for the project. I decided to punt on the question of what to use for the audio/video/graphics programming, and to use Java for the Pascal interpreter. While I was concerned about Java's performance, I had heard that recent versions had substantially improved in this regard, so Java would probably be at least good enough to run the interpreter and control some external programs if necessary for efficiency. Java also offered some great advantages if usable:

- It is a very popular language, and so is likely to remain supported for the foreseeable future – an important consideration for a preservation project.
- It is freely available for most computer systems.
- It avoids a large group of memory bugs and leaks possible in C/C++.

The interpreter went fairly smoothly – I began by writing a recursive descent parser, which was fairly easy to do step by step in Java. That is, I began by parsing the tokens only, and then one by one added the standard Pascal statements and expressions. When I was done, what remained when I ran the parser on the authoring system was the nonstandard code used in the original Erl-King, which was therefore easy to pick out. I also used this strategy later on to find and make sense of the external device control code in the runtime system; this would probably have been harder to do using an object-code level emulator.

After the parser came the portion of the interpreter that actually ran the parsed Pascal code. Initially I was still concerned about performance, and how carefully to program for efficiency. After a few back of the envelope calculations, this concern quickly evaporated as I came to the conclusion that the new midrange PC I would use for the update would be a factor of 10,000 or more times faster than the original system:

- The original had a 4 MHz clock (4 million cycles/second) compared to around 3 GHz for the new PC, for an initial factor of around 800 times faster.

- The original processor required multiple cycles to do many instructions, where the new PC can frequently combine multiple instructions into one cycle – call this another factor of 5.
- The original processor's instructions worked on 8 or 16 bits at a time, compared to up to 64 bits for the new PC. At this point I stopped counting!

I therefore programmed this portion of the interpreter with absolutely no regard for efficiency, and in fact it still wound up being too fast. That is, the only speed adjustments I later had to make in the runtime system were to slow some things down to match the original system performance.

Programming the interpreter went smoothly, and by the first week in December I had a running version of the authoring system. Reassuringly, this only required a few days of debugging and was able to edit data files transferred from the original CP/M system.

Second stage: Runtime system and video player

After the successful update of the authoring system, my initial plan had been to move on to the video player to be used by the runtime system. As usual, however, some problems came up. A combination of family visits and delays in finalizing the contract and receiving the captured video conspired to slow down work for about the next month . Fortunately, the exhibit date was also postponed by a month around this time.

When I did get rolling again, the captured video was still not available so I had to make a switch from my original proposal, and tackle the runtime system next, using a dummy video display. Once again, this went fairly smoothly. Since I was working with the source code, it was fairly easy to make sense of the nonstandard parts of the program that were controlling the external presentation devices (laser disk players, etc). Again, this might be harder to do using object code emulation since it's necessary not only to find the control code but also to figure out the format of the control messages and replies from devices. By the start of February, the simulated runtime system was up and running using a dummy video display (frame counters only), and the mouse for touchscreen input.

One roadblock that did come up here was the overlay text and graphics in the original Erl-King. The SMC-70 supported a text and graphics screen which could be superimposed over the playing video by a specialized piece of hardware. The original Erl-King used this to display text messages and a small set of graphics files on one of two playback monitors. Based on the large amount of CP/M information I had already found on the web, I expected to easily find the graphics file format so I could decode and display the files. Many fruitless hours later, I gave up on this idea – the format had somehow vanished into the bit bucket of history. I toyed with some desperate ideas (ex. displaying the graphics files on the original system and digitally photographing them, then displaying the photographs on the new system). Finally I reasoned that the format must be pretty simple if it could be decoded in a reasonable amount of time on the original system, and was eventually able to decipher it by looking at the raw bytes in the graphics file.

Let this be a cautionary tale for those tempted to use some obscure graphics or video format for their current projects – I doubt it would be possible to make sense of today's much more complex formats in this way!

One interesting decision I faced at this time was on the video format and player to be used. Around six years earlier I had worked on a high-end digital film system made by Philips called the Virtual Datacine (VDC). The VDC captured film at high resolution with no compression (up to around 2K by 1.5K pixels, or around 350 megabytes/second). The resulting data was fed in real time into an impressive array of custom electronics that could do various effects and convert to different standards (ex. NTSC, PAL, HD). The head of the project at the time was rather fanatical about image quality (thus the lack of image compression), and strongly suggested that as a video format we simply use a series of uncompressed single-image files. I liked this idea, but it turned out to be undoable on the system we were using (a $500,000 machine from Silicon Graphics).

When I considered the video format to be used for Erl-King update, I remembered the VDC experience and thought that the idea of using single-frame files was even more appealing for archival purposes. Rather than depending on a particular video coding scheme, we could simply use the highest quality image possible (a completely uncompressed bitmap). As a bonus, the image coding and the specification of how to play a video stream then becomes trivial, or at least expressible in a few sentences. I decided to try using this format, and therefore asked that the video be captured as a sequence of uncompressed bitmaps at as high a quality as possible, combined with uncompressed audio files. The total video size would then come to 150 GB for the video material in the original Erl-King, a manageable quantity on a PC.

The next question was how to play the video. I had initially planned to use an external video server or player, controlled from the Java interpreter either on the same computer or over a network. As I looked into the player requirements, it became apparent once again that it would probably be desirable to roll my own video player. One reason for this was the need to switch rapidly and cleanly between playing video streams. That is, in the original system there were three laser disk players that could be playing at once. In response to a viewer's touch, the system could be required to switch the display between the playing disks, immediately and without a visible glitch.

Once again, a recent work experience was relevant to my decision. A couple of years ago I had worked for a company called Visible World. VW that had the somewhat Orwellian plan of producing video advertisements that were customized based on an individual viewer's profile. The video playback for VW also required the ability to cleanly switch among several possible streams of video. We had therefore tried to do this with many of the existing digital video players (ex. Quicktime, Windows Media Player, RealPlayer). After lots of effort we found that it simply wasn't possible to do this with acceptable quality on the existing players, even though some of the player documentation claimed this ability. This was caused by difficulty in synchronizing between streams, as well as unpredictable delays when starting playback. Based on this experience, I thought it would be be safer to program my own video player that would actually read up to three video streams at once, and decide at each frame switch time which stream to display; in this way, I could be sure that the switch would occur cleanly and at any desired time.

There were some other considerations that argued for a custom video player:

- The updated Erl-King would then be self-contained and not depend on external software that would also need to be maintained over time, a benefit from a preservation standpoint.
- The original Erl-King system displayed video on two monitors, a small one with a touchscreen that the viewer sat in front of, and a large one that displayed the identical video for passersby. For technical reasons, the text and graphics overlay appeared only on the small monitor in the original system, and so it was desirable to duplicate this arrangement in the updated system. I thought this might be tricky to do with an existing video player.
- I was unsure about the ability of existing video players to play single-frame files directly as a video stream.

The next question was what language to program the video player in. I thought it might be necessary to use C or C++ for performance, but after looking at the new sound and video capabilities that had been introduced in recent versions of Java I concluded that it would be worth a try to write the player entirely in Java. If this worked, it would have the benefit of making the updated Erl-King entirely written in a freely available computer language and independent of system-specific software and API's.

The video material for the original Erl-King consisted of three reels, each of which was played off one of the laser disk players in the running system. I had asked for the first reel to be captured separately to test whether the playback would work. This material arrived sometime in early February, and I was able to program the Java player shortly afterward and verify that it could keep up on playback and maintain audio synch. This required some experimentation to find a Java graphics mode fast enough to both display the video images and superimpose the graphics/text. In order to keep up on playback on the completed system, I had to install two extra hard drives and put the video material for each reel on a separate drive. The video looked pretty good to me, though the colors were perhaps a little too saturated. By Feb. 18, the player was working well and things seemed comfortably on schedule.

Once again some problems now came up, this time of a technical nature. The most vexing one of these was the frame numbering scheme used in the original Erl-King to allow jumping to various points within the video material. The material had originally been shot on film and transferred to video using a 3-2 pulldown, with the result that some of the frames were a mix between two different images. In order to avoid jumping to a mixed frame, the laser disks contained a frame numbering that assigned consecutive numbers to the clean (unmixed) frames only. Upon examination, it became clear that this mapping was not perfect; frame numbers were occasionally dropped or duplicated, and there was no known way to read the numbering automatically. It therefore was necessary to map the frame numbers by hand, checking against the original material on the laser disks. I wrote a small application to help in the mapping process, but this was still rather slow and tedious. Fortunately I had a lot of help in mapping, and wound up only doing part of one reel myself.

A related decision we made was to use 24 frame per second playback, in effect reversing the film-to-video process and displaying the original film frames. This was possible with a little extra effort in the mapping process, and in my opinion lead to a better looking display. There was some controversy about whether this “improvement” was in fact desirable. Ultimately, we decided in consultation with the artist that displaying the film frames at 24 FPS was the best choice. This was implemented by adding a preprocessing step converting the video images to film images based on the maps. As a side benefit, it also allowed inclusion of other desired processing such as cropping the images or applying filters.

Due to these and some other minor technical issues, I didn’t have the full updated system running until early March. This lead to a rather frantic week or two of debugging; unlike the authoring system, the runtime system presented some more subtle bugs. There were a couple of deadlock issues caused by conflicts between the concurrently running player and interpreter. There was also some tuning necessary to get the required playback performance, including installing extra drives, placing each processed video reel on a different drive, and programming recovery modes for unexpected delays in disk reading. Finally, there was a considerable amount of timing adjustment necessary to approximate the response times in the original system. In the end, none of these problems proved insurmountable and the full system was ready for installing on March 14. Over the weeks since it has been pretty reliable; it’s been in everyday, almost continuous use with no reported failures or major problems.

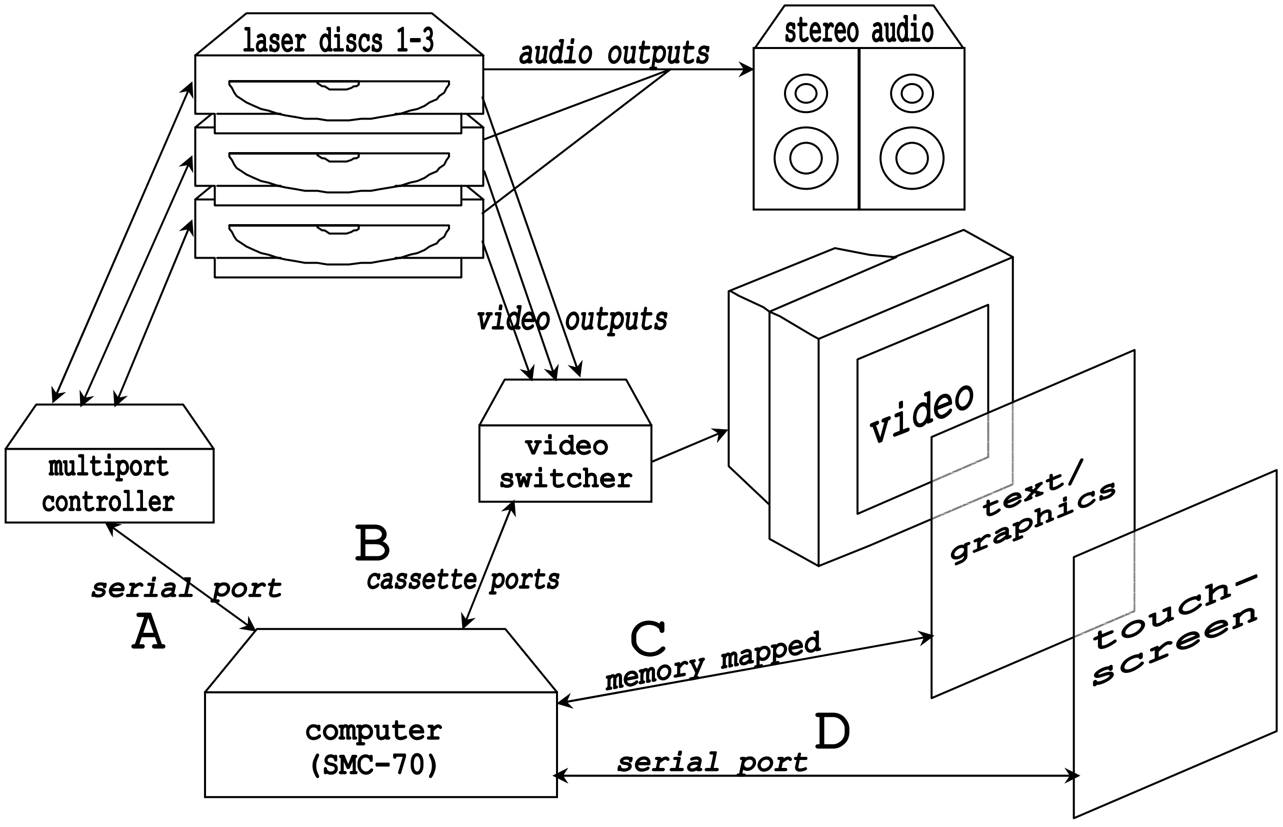
Conclusions

The experience of updating the Erl-King has been interesting in several ways. For video and film projects in particular, I think the use of Java for display and uncompressed single frame images for archiving and preservation offers an interesting combination of high quality and low cost. I was surprised to find that this approach is now feasible on a reasonably capable PC, and of course the quality and cost should only improve over time.

I also think that some more general lessons can be drawn from this project about the uses of emulation in this type of preservation effort. I would summarize them under the heading of “no silver bullet”. That is, in computer science it’s always tempting to overstate the benefits of a single technique, be it emulation, object oriented programming, networked systems, etc. In practice you usually find that the technique has some benefit but that most of the devil is still in the details of the particular project you’re working on.

In the case of preservation through emulation, many of the devilish details that cropped up here are likely to recur on similar projects. Most artworks that include computers of this period are likely to also use a substantial amount of external hardware that must be emulated, since the computers were not very able at video or graphics. The emulation of these external devices is likely to be a large part of the entire programming effort. The control interfaces used for the devices will also need to be deciphered and debugged, which may be difficult with a pure object-code emulation approach. The original works will also probably contain some subtle dependencies on system behavior upon program errors, which the emulator must match. At the least, the last two problems will probably require significant work in customizing the emulator for the particular project.

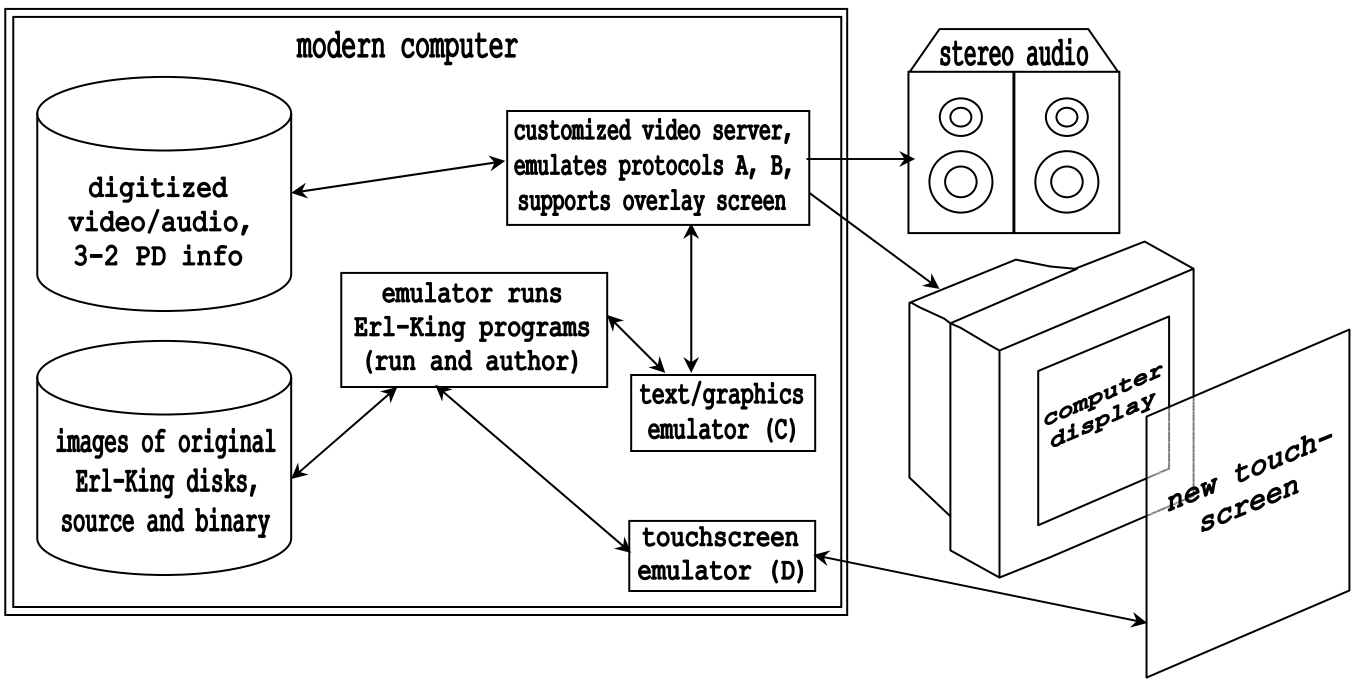
To sum up, I would advocate a pragmatic development approach that recognizes that each preservation project will require its own set of techniques and compromises. Since some of these will not be anticipated in advance, I think the best process for this type of project would as a rule be a relatively free-flowing one involving developers, conservators, and if possible the original artists. For those familiar with software development jargon, this would be in the "extreme programming" category (a flexible iterative process using partial releases and repeated feedback) as opposed to the "waterfall" approach (a more rigidly planned process).



original system

A-D are protocols to external devices

proposed new system


modern computer
digitized video/audio, 3-2 PD info
customized video server, emulates protocols A, B, supports overlay screen
stereo audio
emulator runs Erl-King programs (run and author)
text/graphics emulator (C)
computer display
images of original Erl-King disks, source and binary
new touch-screen
touchscreen emulator (D)