Vivio – A System for Creating Interactive Reversible E-Learning Animations for the WWW

Dr. Jeremy Jones
Dept. Computer Science, Trinity College,
Dublin 2, Ireland

jones@cs.tcd.ie

Abstract

Vivio is a system that makes it easier to create interactive reversible 2D vector based E-learning animations for the WWW. Vivio programs describe how the properties of graphical objects change as function of time or as the result of external events. Since Vivio animations follow the execution of a program, they can respond to user input and are not limited to preset animation sequences. A key feature of the Vivio player is its ability to play Vivio animations smoothly in both forward and reverse directions.

1. Introduction

A 2D animation is often the best way to illustrate a complex idea. Animations have a broad applicability in many subject areas, such as Computer Science, but in order for such animations to be widely available, it is important that the animations themselves can be created in a timely manner, without excessive computer science skills.

A good educational animation should be easy to use, scale with the window in which it is displayed, animate smoothly and should be interactive so as to encourage explorative self-directed learning. Animations that are too jerky or have too much concurrent movement make it very difficult to focus on what is really happening. Furthermore, the animation speed should be adjustable and it should be possible to single-step the animation in both directions and quickly snap forwards and backwards to key frames.

The objective of this work is to make it easier to create such high quality E-learning animations for the WWW. This is achieved by providing a programming model with an appropriate level of abstraction and an efficient runtime that handles much of the low-level detail such as repainting the screen and playing animations backwards.

2. Related Work

Most of the animations developed with Vivio, so far, are best described as concept animations [1] rather than algorithm animations [2]. Although systems such as Java [3] and Flash [4] are used to develop animations, their programming models are either at too low a level or not powerful enough to create good educational animations without considerable effort.

Few systems are able to execute programs backwards, as needed to play Vivio animations in reverse. ZStep95 [5] approaches reversible execution from the standpoint of program debugging whereas LEONARDO [6] is a system for animating C programs that supports fully reversible execution. This is achieved by using a virtual machine where the execution of each virtual instruction is reversible. Enough state is saved when each virtual instruction is executed so that its operation can be undone. This approach not only saves a lot of state, but also slows forward execution and makes it difficult to snap forwards and backwards quickly.

Vivio animations step from one frame to another, with the execution of many "instructions" in-between frames. Consequently, Vivio uses an incremental state saving approach where the difference in program state is saved every *n* frames [7]. As multiple changes to an object can occur between state saves, less state is saved overall. Code must be re-executed, however, to get from a saved state to a particular frame. The frequency of state saves can be set dynamically to trade memory usage for saving state against reverse animation speed.

3. Programming Model

Although the syntax, semantics and functionality of the Vivio programming language are similar to C++ and Java, the Vivio runtime maintains a list of events ordered by time

Proceedings of the 2004 IEEE Symposium on Visual Languages and Human Centric Computing (VLHCC'04)
0-7803-8696-5/04 $ 20.00 IEEE

measured in ticks. Animations progress by incrementing the current tick in real time. On each tick, the player takes those events from event Q that match the current tick, executes the code pointed to by the event PC and then redraws the screen. External events such as mouse clicks and key presses are handled by associating code with the event and when such events occur, an event is effectively added to the front of the event Q for immediate execution.

Graphical objects are created in a user virtual co-ordinate space that is mapped onto the display window using the setviewport(…) function. For example, the following expression creates a rectangle graphical object:

r = Rectangle(*layer*, *group*, *quality*, *pen*, *brush*, *x*, *y*, *w*, *h, txtbrush*, *txtfont*, *txt*);

The properties of graphical objects can be changed using animated functions such as:

r.translate(*dx, dy, n*, *ntick,*, *wait*);

The rectangle is moved *dx, dy* units in *n* equal steps with *ntick* ticks between each step (N.B. if *n* = 0 the translation is effected immediately). This is implemented by adding an event to the event Q at *tick* + *ntick* to move the rectangle one step, which when executed will add another event at *tick* + *ntick* until all *n* steps are executed. If *wait* = 1, forward execution is suspended until the function is complete. Although animated functions provide a simple mechanism to express concurrency, a fork statement is also needed to create additional "threads" by adding an event to the front of the event Q to call a user defined function.

4. The Player

Vivio is written in Visual C++ and runs on Windows based PCs. A fast single pass compiler generates compressed vcode which is fetched by the player (normally hosted in a web page), JIT compiled into Intel x86 machine code and then executed.

Users quickly become dissatisfied if animations are played too slowly, so rendering speed is critical. All graphical objects changed since the last render are flagged and their original and new minimum bounding boxes (mbbs) calculated. Only objects that overlap these mbbs are redrawn. Layered memory based bitmaps may also be used to accelerate rendering. Note that the render times for stepping forward or back one tick should be identical as the same area of the display is redrawn in each case.

In order to play an animation backwards, it is "reset" and "played" forward as fast as possible without updating the display until tick-1 is reached, then tick-2 and so on. To accelerate this process, the player automatically takes incremental snapshots of the program state. A snapshot is always taken at tick = 0 after all initialization code has executed and the first frame rendered. Further snapshots are taken ONLY when playing backwards, either at specified intervals or whenever the "execution" time from the previous snapshot exceeds some threshold.

To reduce the render time when playing backwards, the current render state of all graphical objects is saved. Before a graphical object is rendered, a check is made to see if it matches its saved render state. This optimisation is important for efficient animation in the reverse direction.

All user actions (e.g. mouse clicks and key presses) are also saved on an asynchronous event Q that persists when the animation state is restored. These asynchronous events are replayed unless the user interacts with the animation, in which case they are discarded as the animation is considered to have taken a new path.

5. An Example Vivio Animation

Consider a typical interactive Vivio E-learning animation that illustrates the operation of the write-once cache coherency protocol [8]. Cache coherence is a core topic in the IEEE/ACM Computing Curricula 2001 (section AR7).

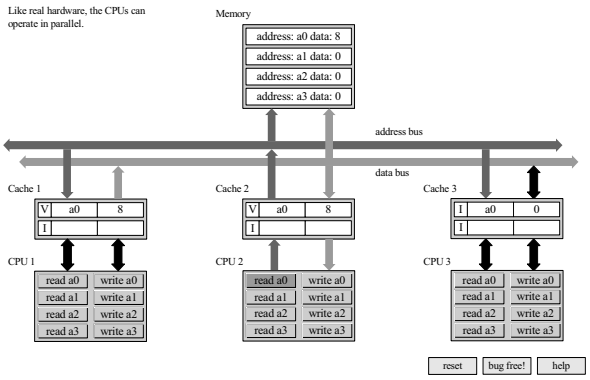

Figure 1. Write-Once Animation

The animation is created in a virtual 800 x 600 pixel window. A multiprocessor system is depicted with main memory and three CPUs, each with their own direct mapped local cache (Fig. 1). Main memory comprises four locations a0 to a3. Even addresses map onto cache set 0 and odd addresses onto set 1. The protocol state of each cache line is indicated by the letters I(nvalid), V(alid), R(eserved) and D(irty). Each CPU contains buttons that initiate a read or write transaction to the specified memory location. Blue and red animated arrows indicate the flow of

traffic on the address and data busses respectively. The cache lines and memory location involved in a transaction are coloured green and the stale memory locations gray.

The reset button at the bottom right hand corner is used to reset the animation. The bug button introduces bugs into the protocol and challenges users to find out exactly what they are. The help button is a link to an HTML help page.

Like a real multiprocessor system, the animation allows transactions to be performed concurrently by the different CPUs. For example, all CPUs can perform cache reads simultaneously. If more than one CPU needs to perform a bus transaction, however, access to the bus is serialized by simulating a "bus lock" within the Vivio code.

Since the animation can be single stepped forwards and backwards using the mouse wheel, great care must be taken with the details of each animation step. The ability to single step and replay transactions makes the animation far easier to follow, understand and use. The size of the compressed vcode for this animation is less than 3K.

6. Performance

Performance statistics for a number of sample animations were collected using a 1.13GHz DELL Inspiron 8100 with 256K DRAM. All animations are displayed in an 800x600 32 bits per pixel window and animated at 100 ticks/sec. The DLX/MIPS animation is a configurable register transfer level simulation of a DLX/MIPS CPU that illustrates a number of pipelining techniques [9].

Table 1.

	playing forwards			playing backwards		
	# frames	avg exe (ms)	avg render (ms)	# frames	avg exe (ms)	avg render (ms)
ExchangeSort	2,993	0.004	4.24	2,990	0.80	4.16
WriteOnce	995	0.009	9.36	963	2.35	9.26
DLX/MIPS	3,362	0.019	4.21	3,360	5.46	4.41

Table 1 shows how much time is spent "executing" and rendering each animation while playing forwards and backwards. The results show that 2 to 3 orders of magnitude more CPU time is spent rendering than "executing" when playing forwards. Note the increase in "execution" time when playing in reverse and that the average forwards and backwards render times are approximately equal. In some cases, the sum of the "execution" and render times exceeds 10ms, the target for rendering at 100 fps. Taking snapshots more frequently or using a faster CPU will reduce these times.

7. Conclusions

A brief overview of the Vivio system has been given and although an experimental system, it has allowed a useful collection of interactive Computer Science E-learning animations to be developed [10]. Feedback indicates that these animations have helped students to better understand complex topics such as sorting, cache coherency protocols and pipelining.

8. Acknowledgements

I would like to thank the many students and colleagues at TCD who have helped with the development of Vivio over the years.

9. References

[1] Birch, M.R. et al., "DYNALAB: A Dynamic Computer Science Laboratory Infrastructure Featuring Program Animation", ACM SIGCSE Bulletin 27(1) pp 29-33, 1995.

[2] Kerren, A. and Stasko J.T., "Algorithm Animation", Software Visualization 2001 pp 1-15, 2001.

[3] Gosling, J., Joy, B., Steele, G. and Bracha, G. "The Java Language Specification – Second Edition", Addison-Wesley 2000.

[4] www.macromedia.com/software/flash

[5] Lieberman, H. and Fry, C., "ZStep95: A Reversible, Animated Source Code Stepper", in Stasko, J.T. et al., editors, Software Visualization: Programming as a Multimedia Experience, pp 277-292, 1998.

[6] Crescenzi, P., Demetrescu, C., Finocchi, I., and Petreschi, R., "Reversible execution and visualization of programs with LEONARDO", Journal of Visual Languages and Computing, 11(2) pp125-150, 2000.

[7] West, D. and Panesar, K.S., "Automatic Incremental State Saving", Proceedings of the Tenth Workshop on Parallel and Distributed Simulation, pp 78-85, 1996.

[8] Goodman, J.R. "Using Cache Memory To Reduce ProcessorMemory Traffic", Proceedings of the 10th Annual International Symposium on Computer Architecture, pp 124-131, June 1983

[9] Hennessy, J. and Patterson D.A. Computer Architecture A Quantitative Approach, Morgan Kaufmann1990.

[10] www.cs.tcd.ie/Jeremy.Jones/vivio/vivio.ht