Abstract

Vivio is a system that makes it easier to create interactive reversible 2D vector based e-learning animations for the WWW. Vivio animations describe how the properties of graphical objects change as a 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, consequently, not limited to preset animation sequences. Vivio animations are compiled into a compact intermediate code which is normally executed by the Vivio player hosted in a web page. A key feature of the Vivio player is that it can 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 can be extremely useful in helping Computer Science students, for example, understand diverse topics such as sorting algorithms, tree algorithms, network protocols, cache coherency protocols and processor pipelines. Clearly, animations have a broad applicability in many subject areas, 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.

Good educational animations should be easy to install and use, scale with the window in which they are displayed and animate smoothly and with purpose. Too fast, jerky or too much concurrent movement makes it difficult for the eye to focus on what is really happening, although sometimes this may be unavoidable by the very nature of what is being animated. The animation speed should be controllable and it should be possible to single-step in both forward and reverse directions or quickly snap forwards and backwards to key points in the animation. Furthermore, animations shouldn't be passive, always following the same fixed path. They should be interactive, responding to user actions, so as to encourage explorative self-directed learning.

The objective of this work is to make it easier to create high quality E-learning animations for the WWW. This is achieved by providing a programming model with an appropriate level of abstraction and by providing a runtime that handles much of the low-level detail such as repainting the screen efficiently and playing animations backwards. A second objective is to make detailed performance measurements to test how well the animations run on standard PC hardware (i.e. that typically used by students).

2. Related Work

Most of the animations developed with Vivio could be best described as concept animations [1] rather than program or algorithm animations [2], the emphasis being on the high level view rather than trying to relate every animation step to a line of source code. Although systems such as Java [3], Flash [4] and W3C scalable vector graphics [5] are used to develop animations, their programming models are at too low a level or not powerful enough to create good educational animations, as described above, without considerable effort, if at all.

Few systems are able to execute programs backwards, as is needed to play programmed animations in reverse. A number of systems approach reversible execution from the standpoint of program debugging. ZStep95 [6] is a reversible and animated source code stepper for LISP programs. LEONARDO [7] 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. A similar system for reverse
execution of Java bytecodes is described in [8]. This approach saves a lot of state, not only slowing forward execution, but also making it difficult to snap forwards and backwards quickly. Another approach is to automatically generate a reverse program that is able to undo the effects of forward execution [9]. This technique saves less state, as the reverse program can regenerate some of the restored state dynamically.

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 state is saved every n frames [10]. As multiple changes to an object can occur between frames, less state needs to be saved overall. Code must be re-executed, however, to get from a saved state to a particular frame, but since the Vivio runtime is based on a JIT compiler this is quick compared with an interpreter. The frequency of state saves can be adjusted to trade memory usage against reverse animation speed.

3. Event Based Animation

A Vivio animation comprises a collection of graphical objects that change appearance as a function of time or as the result of external events. The Vivio player maintains a list of events ordered by time measured in ticks (Fig. 1).

![Event Q Diagram](image)

Animations progress by incrementing the current tick in real time (e.g. 20ms per tick). The player takes all events that match the current tick, in turn, from the head of the event Q, executes the code pointed to by the PC and then redraws the screen. This continues until the event Q is empty. Note that the executed code may add events to the event Q. External events such as mouse clicks and key presses are handled by associating code with the event. When such an event occurs, an event is effectively added to the front of the event Q for immediate execution.

The amount of time spent redrawing the screen must be minimised and standard techniques such as double buffering employed so that the display is updated smoothly. CRT and TFT LCD displays are refreshed at between 50 and 100 times per second and, consequently, there is benefit in redrawing the screen at a faster rate.

4. Vivio Basics

The syntax, semantics and functionality of the Vivio programming language are similar to languages like C++ or Java and consequently only the special features pertaining to animations are described. Vivio supports integer, real and string types, functions, objects, methods, arrays and associative arrays and a standard set of control statements.

An animation is built up from a number of layers ordered in z-order. By default there is a single background layer with z = 0. When graphical objects (rectangles, ellipses, lines, polygons, arcs, pies, Beziers, splines, images and shapes) are created, they are created on a particular layer. Layers are always drawn in ascending z-order and the graphical objects belonging to a layer are drawn in creation order. Functions are available to change the ordering and the layer to which a graphical object belongs.

Graphical objects may optionally belong to a group. Altering the property of a group is a convenient way to change the property of all objects in that group.

Graphical objects are created with co-ordinates defined in a user virtual co-ordinate space. The virtual window that is mapped to the display window can be set using the setviewport(...) function:

```
setviewport(x, y, w, h, keepaspect)
```

The virtual window positioned at x, y, with width w and height h is mapped to the display window (normally a window in a web page). The keepaspect parameter determines if the mapping preserves the user aspect ratio. Setting a viewport means that the animation will scale with the size of the display window.

Graphical objects are drawn using pens, brushes and fonts. Pens are used to draw lines and borders, brushes to fill graphical objects and fonts to draw text. The properties of existing pens, brushes and fonts can be changed dynamically which has the effect of causing all objects drawn with them to be redrawn.

```
pen = SolidPen(style, width, colour); // create pen
brush = SolidBrush(colour); // create brush
font = Font(fontname, height, flags); // create font
pen.SolidPen(style, width, colour); // change properties
brush.SolidBrush(newcolour); // change properties
font.Font(fontname, height, flags); // change properties
```

The opacity of a graphical object is specified as a value in the range 0 (invisible) to 100 (opaque). The default opacity for a newly created graphical object is 100.

Consider the following expression to create a rectangle
graphical object:

\[ r = \text{Rectangle}(\text{layer, group, quality, pen, brush, } x, y, w, h, \text{txtbrush, txtfont, txt}); \]

![an example text string](image)

Figure 2. Rectangle Object

A rectangle with top left corner at \( x, y \), width \( w \) and height \( h \) in user co-ordinates is created (Fig. 2). The \( txt \) string is displayed in the middle of the rectangle (by default) using \( \text{txtbrush} \) and \( \text{txtfont} \). A border is drawn with the \( \text{pen} \), the graphical object filled with the \( \text{brush} \) and the \( \text{quality} \) parameter determines whether the graphical object is drawn with or without anti-aliasing.

Graphical objects represented by a number of points (e.g., lines, polygons, Beziers and splines) are created using functions which accept an arbitrary number of co-ordinates. For example, the following statement creates a triangle:

\[ p = \text{Polygon}(\text{layer, group, quality, pen, brush, } x_0, y_0, x_1, y_1, x_2, y_2); \]

Functions are available to change the properties of a graphical object such as its position, size, opacity, the number of points and the co-ordinates of a particular point:

\[
\begin{align*}
  r &. \text{translate}(dx, dy); & \text{set position relative} \\
  r &. \text{setsize}(w, h); & \text{set size} \\
  r &. \text{setopacity}(\text{opacity}); & \text{set opacity} \\
  p &. \text{setnpoints}(n); & \text{set number of points} \\
  p &. \text{setpoint}(n, x, y); & \text{change point } n
\end{align*}
\]

The initial co-ordinates of a graphical object are specified on creation. A drawing editor metaphor is used whereby each graphical object is considered to be enclosed by an imaginary minimum bounding box (mbb). Functions such as \( \text{setpos(...)} \) and \( \text{setsize(...)} \), set the position and size of the imaginary mbb and then transforms the original graphical object to fit.

5. Animating Graphical Objects

In order to create an animation, the properties of graphical objects need to change as a function of time and changes may need to be applied to a number of graphical objects in parallel. Consider the following function to move a rectangle \( r \) horizontally:

\[
\text{function move(Rectangle } r) \\
  \text{for (int } t = 0; t < 10; t++) \\
  r.\text{translate}(1, 0); \\
  \text{wait}(1); \\
end;
end;
\]

The \( \text{wait}(n) \) function causes execution to be suspended for \( n \) clock ticks. Internally, an event is created and added to the event \( Q \) at \( \text{tick} + n \) with the execution state (stack frames and PC) saved in the event (Fig. 3). After the specified number of clock ticks, the event state is restored and execution resumed. Note that globally accessible data can be changed by the execution of other events in the meantime.

![Figure 3. Runtime Stack](image)

Now consider the following code for moving two rectangles \( r_0 \) and \( r_1 \) concurrently:

move(r0); // move r0 and then…
move(r1); // move r1

As it stands, \( r_0 \) and \( r_1 \) will not move concurrently – \( r_0 \) will move first followed by \( r_1 \). For concurrent movement, the code needs to be modified to create another "thread" as follows:

fork(move(r0)); // separate "thread" to move \( r_0 \)
move(r1); // move \( r_1 \)

… \( \text{following statement} \)

In this case, \( r_0 \) and \( r_1 \) will move concurrently. \( \text{fork}(...) \) results in an event being added to the head of the event \( Q \) with its PC initialised to point to the code for executing the function parameter. This creates the illusion that a separate "thread" exists for moving \( r_0 \).
Fork(…) is seldom needed, however, to provide concurrency as most functions that change the properties of graphical objects, pens, brushes and fonts exist in an animated form. Consider, again, the code for moving rectangle r horizontally:

```java
r.translate(10, 0, n, ntick, wait);
```

Rectangle r is moved 10 units horizontally. The translation is performed in n equal steps with ntilec ticks between each step. The first step is performed at tick + ntilec and the last at tick + n*ntec. The function is executed by adding an event to the event Q at tick + ntilec, which when executed will add another event at tick + ntilec until all n steps have been executed. If the wait parameter is 0, execution continues immediately with the following statement, otherwise forward execution is suspended until the translate function is complete. If the n, ntilec and wait parameters are not specified, they default to 0 resulting in the function being executed immediately. The following code uses this approach to move rectangles r0 and r1 concurrently:

```java
r0.translate(10, 0, 10, 1, 0); // DON'T wait
r1.translate(10, 0, 10, 1, 1); // wait
… // following statement
```

Each rectangle is moved horizontally 10 units in 10 equal steps with 1 tick between each step. Since each translation takes the same number of ticks, the following statement is executed when both functions are complete. Animated functions are implemented far more efficiently than explicit forks and, consequently, should be used in preference whenever possible.

Functions to change the properties of graphical objects, pens, brushes and fonts exist in animated versions where it makes sense. The setpoint(n, x, y, n, ntilec, wait) function, for example, will animate the specified point from its current position to the specified position.

### 6. User Event Handling

User events (mouse and keyboard) are handled by using "when" statements to associate code with the particular event. The following code, for example, attaches a "left mouse button" event handler to a graphical object r:

```java
when r.eventLB (int down, int x, int y) …
end;
```

The body of the when statement is executed if the left mouse button is pressed or released when the mouse is over graphical object r. The down parameter indicates whether the button is being pressed or released and the x and y parameters give the mouse position in user co-ordinates. The body of the when statement is effectively executed as a function call. A graphical object can have more than one event handler for a particular event.

If the mouse or keyboard is over more than one graphical object, the event handlers are executed in "front to back" order (i.e. the reverse of the drawing order). If an event handler doesn't want to pass the event up to the event hierarchy it can return 1 (by default it returns 0).

A graphical object can grab (or ungrab) all mouse and keyboard events by calling r.grab() (or r.ungrab()). Only one graphical object can grab events at a time, in which case all mouse and keyboard events are sent to this graphical object before being passed up the event hierarchy.

An event handler can be executed when any property of a graphical object is changed using the eventUPDATED event.

### 7. The Player

The implementation, described here, is written in Visual C++, uses the gdi+ graphics library and runs on Windows based PCs. A fast single pass compiler, with simple peephole optimization, is used to generate compressed vcode. A Vivio ActiveX player retrieves the vcode and JIT (just in time) compiles it into Intel x86 code for execution. The compiler and vcode organization takes much from [11].

![Figure 4. Vcode formats](image)

Each vcode instruction comprises one, two or three 16-bit words and is encoded in one of 5 instruction formats (Fig 4). The 00 format contains an 8 bit opcode and 8 bit operand. If the operand is larger than 8 bits, the instruction is encoded using the FE or FF format (NB: op remains the same). Vcode instructions that do not need an operand use the FC format and those that accept an optional number of
parameters use the FD format (the n field encodes the actual number of parameters pushed onto the stack).

Vcode is a single address instruction set designed for an accumulator based stack machine. A statement such as "i = j + k*m" results in the following instruction sequence:

LD  j // load accumulator with j
PL  k // push&load with k
MUL m // multiply accumulator by m
ADD // pop and add to accumulator
ST  i // store accumulator in i

Push&load means push the accumulator onto stack and then load the accumulator. This style of instruction set results in less instructions and more compact code than a pure stack machine.

The "virtual" machine contains an accumulator (ACC) and three registers G, OBJ and FP. G points to the global variables, OBJ to the variables belonging to the active object and FP to the local variables in the current stack frame. Variables are accessed relative to one of these registers and consequently vcode supports 4 addressing modes:

LD 123 // immediate - load immediate 123
LD G[1] // global - load global variable 1

The vcode instruction set reduces the so-called "semantic gap" by directly supporting the functionality of Vivio. There are vcode instructions, for example, for creating and altering the properties of all the supported pen, brush, font and graphical objects. There are also many instructions to push (& load) single and multiple constants as this has been found to reduce considerably the size of the generated vcode.

To play a Vivio animation, the player needs to "execute" the corresponding vcode. This is achieved by JIT compiling the vcode into x86 machine code and then executing it. The player performs the JIT step with a quick single pass through the vcode. It makes sense to expend more effort in the compilation phase, performing optimisations and collecting metadata at the vcode level, as this step is performed once whereas the JIT phase is possibly performed each time the program is executed. The collected metadata assists the JIT phase to produce better quality code in less time.

Vcode may also be interpreted, but with some loss of performance. A Java based interpreter has been written to interpret vcode, but the strict Java type system impedes fast execution (at least a 100 times slower than the implementation described here). The advantage of the Java interpreter, however, is its portability.

The JIT generated code uses the x86 registers eax, edx, esi and edi for the vcode ACC, G, OBJ and FP registers respectively. Complex vcode instructions are implemented directly in C++ and are executed by generating a call to a corresponding C++ linkage function.

The JIT generated code is executed by calling the execute(...) function. Vcode builds up its own stack frames on the C++ stack as it executes. When the execution of a "thread" is suspended (e.g. wait(n)), the state of the vcode stack is saved in an event on the event Q. The size of the vcode stack is normally small. The vcode stack frames are removed from the C++ stack before the execute(...) function returns. When an event is executed, any associated state saved in the event structure is reconstructed on the stack before execution starts.

Users quickly become dissatisfied if animations are played too slowly and, consequently, the rendering speed is critical. All graphical objects changed since the last render are flagged and their original and new minimum bounding boxes (mbbs) calculated. All objects that overlap these mbbs need to be redrawn (clipped to the corresponding mbb) in drawing order. It is costly to maintain unlimited mbbs for this purpose and so Vivio maintains 8 mbbs by default for each layer, but allows the programmer to set the number dynamically at runtime. It should be noted, however, that the render time for stepping the animation forward or back one tick should be identical as the same area of the display is redrawn in each case.

Memory based bitmaps are used to speed up the rendering step. Consider the simple case of an animation comprising a fixed complex background with a graphical object moving in the foreground. When the graphical object moves, the images behind its original and new positions need to be redrawn. The images behind the graphical object can be quickly restored if they are blt-ed from a bitmap rather than rendered using drawing primitives. Each Vivio layer can optionally have a cached bitmap and/or a summed bitmap associated with it. The following statement creates a new layer:

```csharp
layerz = Layer(z, flags, transparentcolour);
```

The z parameter specifies the layer z ordering, flags specifies whether the layer has a cached and/or summed bitmap and transparentcolour specifies the transparent colour used as the background for a cached bitmap. A transparentBlt doesn't copy those pixels that match the transparent colour.
Consider the organization of the three-layer animation shown in Fig. 5. All layers have summed bitmaps and layer 1 has a cached bitmap as well. If the triangle is moved, the area covering its old and new positions can be blted from the summed bitmap (5), the triangle rendered (6) and then the final image blted to the display (7). If the circle is moved steps (2), (3), (4), (5), (6) and (7) need to be performed and if the square is moved steps (1), (2), (4), (5), (6) and (7). The optimum use of cached of summed bitmaps is determined by the nature of the animation.

A snapshot contains the program state and includes the state of all global variables, strings, fonts, pens, brushes, graphical objects, objects, arrays, events and event handlers. Typically many objects don't change between snapshots and so it makes sense to save the differences rather than the complete state each time.

8. Playing Animations Backwards

A key feature of the player is its ability to play animations smoothly in both forward and reverse directions. Animations are stepped forward and backward using the mouse wheel or keyboard.

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 speed up this process, the player automatically takes regular snapshots of the program state. A snapshot is always taken at tick = 0 which is the state after all initialization code has executed and the first frame rendered. Further snapshots are taken ONLY when playing backwards, either at regular intervals (e.g. every 1000 ticks) or whenever the "execution" time from the previous snapshot exceeds some threshold (e.g. 10ms). This means that the animation need never be "reset", but can be restored to the state at tick = 0 obviating the need to re-execute the initialisation code to generate the first frame which is often time-consuming.

In order to reduce the render time when playing animations backwards, the current render state of all graphical objects is saved unless unchanged. Before a graphical object is rendered, a check is made to see if it matches its saved render state and if so there is no need to re-render. This optimisation is extremely 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 is restored to a previous state. These asynchronous events are replayed unless the user interacts with the animation, in which case all future asynchronous events are discarded as the animation is considered to have taken a new path.

The basic mechanism described here is also used to snap between checkpoints (key frames) in both forwards and reverse directions, again using the mouse wheel or keyboard. The player remembers the current tick whenever the checkpoint() function is executed.
9. An Example Vivio Animation

Consider a typical interactive Vivio e-learning animation that illustrates the operation of the write-once cache coherency protocol [12]. Cache coherence is a core topic in the IEEE/ACM Computing Curricula 2001 (section AR7).

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 local cache (Fig. 7). Main memory comprises four locations a0 to a3. Each cache is direct mapped with two sets. Even addresses (a0 and a2) map onto set 0 and odd addresses onto set 1. The state of each cache line is indicated by the letters I(nvalid), V(alid), R(eserved) and D(irty) according to the protocol. Each CPU contains buttons that initiate a read or write transaction to the specified memory location. Animated blue and red 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. Stale memory locations are coloured 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.

The animation snapshot shows CPU2 reading memory location a0 (Fig. 7). Since a0 is not in CPU2's cache, CPU2 tries to read a0 from memory. As Cache1 has the only up to date copy of a0 (was in the Dirty state), Cache1 intervenes and drives the data bus with the contents of a0 from its cache. Cache2 and main memory are updated.

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

Since the animation can be single stepped, care needs to be taken with the graphical details of each animation step. The ability to single step the animation forwards and backwards and to replay transactions makes the animation far easier to follow, understand and use.

10. Performance

The Vivio integrated development environment has been instrumented to collect real-time performance statistics using the Intel x86 hardware time stamp counter. All reported measurements were collected using a DELL Inspiron 8100 laptop containing a 1.13GHz mobile Pentium III CPU with 256k DRAM. With this particular CPU, it is important that the Intel SpeedStep technology doesn't reduce the clock speed during measurement. All animations are displayed in an 800x600 32 bits per pixel window and run at 100 ticks/sec.

Table 1.

<table>
<thead>
<tr>
<th>Animation</th>
<th>Compile time (ms)</th>
<th>JIT time (ms)</th>
<th>Compressed vcode size (bytes)</th>
<th>x86 size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExchangeSort</td>
<td>16.57</td>
<td>0.29</td>
<td>1,907</td>
<td>6,356</td>
</tr>
<tr>
<td>WriteOnce</td>
<td>17.98</td>
<td>0.47</td>
<td>2,665</td>
<td>9,510</td>
</tr>
<tr>
<td>DLX/MIPS</td>
<td>43.10</td>
<td>1.68</td>
<td>11,448</td>
<td>41,187</td>
</tr>
</tbody>
</table>

Table 1 compares the compilation time, JIT compilation time and the code sizes for 3 selected Vivio animations. ExchangeSort is an animation of an exchange sort, WriteOnce is an animation of the Write-Once cache coherency protocol and DLX/MIPS is a configurable register transfer level simulation of a DLX/MIPS CPU that demonstrates a number of pipelining techniques [13]. Note the speed of the JIT step and the small sizes of the compressed vcode files.

Table 2.

<table>
<thead>
<tr>
<th>Animation</th>
<th>Playing forwards</th>
<th>Playing backwards</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># frames</td>
<td>avg exe (ms)</td>
</tr>
<tr>
<td>ExchangeSort</td>
<td>2,993</td>
<td>0.004</td>
</tr>
<tr>
<td>WriteOnce</td>
<td>995</td>
<td>0.009</td>
</tr>
<tr>
<td>DLX/MIPS</td>
<td>3,362</td>
<td>0.019</td>
</tr>
</tbody>
</table>

Table 2 shows how much time is spent "executing" and rendering each animation, which is played forwards with a fixed sequence of user actions and then played in reverse back to the start. Results are reported separately for playing
forwards and backwards. The results give the number of frames rendered, the average "execution" time per frame and the average render time per frame. The "execution" time includes any time spent saving and restoring state (more significant when playing in reverse). The results show that 2 to 3 orders of magnitude more CPU time is spent rendering than "executing" when playing forwards. They also show that the average render times when playing forwards and backwards are approximately equal (when playing backwards the initial view at tick = 0 doesn't have to be rendered from scratch). In some cases, the sum of the "execution" and render times exceeds 10ms, which is the target for rendering at 100 frames per second. Taking snapshots more frequently or using a faster CPU will reduce these times.

Table 3.

<table>
<thead>
<tr>
<th>Playing Backwards</th>
<th>avg</th>
<th>avg</th>
<th>avg</th>
<th>avg</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>vcode</td>
<td>save</td>
<td>restore</td>
<td>render</td>
<td>sum</td>
</tr>
<tr>
<td>WriteOnce 1000</td>
<td>2.17</td>
<td>0.001</td>
<td>0.19</td>
<td>9.26</td>
<td>11.62</td>
</tr>
<tr>
<td>WriteOnce 500</td>
<td>1.07</td>
<td>0.001</td>
<td>0.19</td>
<td>9.26</td>
<td>10.51</td>
</tr>
<tr>
<td>DLX/MIPS 1000</td>
<td>4.17</td>
<td>0.001</td>
<td>1.29</td>
<td>4.41</td>
<td>9.87</td>
</tr>
<tr>
<td>DLX/MIPS 500</td>
<td>2.18</td>
<td>0.002</td>
<td>1.31</td>
<td>4.42</td>
<td>7.91</td>
</tr>
<tr>
<td>DLX/MIPS 5ms</td>
<td>1.90</td>
<td>0.006</td>
<td>1.32</td>
<td>4.46</td>
<td>7.67</td>
</tr>
</tbody>
</table>

Table 3 shows the breakdown of the "execution" time into its constituent parts (executing vcode, saving state and restoring state) while playing backwards. The qualifiers 1000, 500 and 5ms indicate the snapshot frequency. Increasing the snapshot frequency improves the playback speed as expected, but the render time still dominates.

11. Conclusions

A brief overview of the Vivio system has been given. Although an experimental system, it has allowed a useful collection of high quality interactive Computer Science e-learning animations to be developed [14]. Anecdotal experience and feedback indicate that these animations have helped students to better understand complex topics such as cache coherency protocols and pipelining.

By providing an appropriate level of abstraction and handling much of the low-level detail in the runtime, Vivio makes it far easier to develop high quality E-learning animations for the WWW. Vivio not only generates very compact vcode files for distributing the animations across the Internet, but Vivio animations scale with the display window, are interactive, execute quickly, and animate smoothly in both forward and reverse directions.

12. Acknowledgements

I would like to thank the many Computer Science and Engineering students at TCD who have helped with the development of Vivio over the years and the Centre for Learning Technology TCD and Edsko de Vries for their contribution towards the development of the DLX/MIPS animation.

13. References


[5] www.w3.org/TR/SVG/


