Editor: Mike Potel

Graphics Performance in Rich Internet Applications

Rama C. Hoetzlein
Aalborg University Copenhagen

Imagine an Internet experience in which online data visualization and interaction are fully dynamic and smooth even when displaying large-scale networks. Several rich Internet application (RIA) frameworks are working in this direction. Most browsers have integrated HTML5 and the new canvas elements, pending final specification, and Adobe recently released Flash Player 10.1 with GPU-accelerated video playback. The major RIA players are exploring GPU-accelerated designs while competing to maintain an interactive experience by reducing virtual code execution and CPU rendering time.[1]

So, what's the best RIA framework for developing large-scale, dynamic online data visualizations, applications, or games? To answer this, I developed a test suite to consistently measure raw graphics performance in RIAs. It consists of a simple sprite-based particle system coded and optimized on each framework. By testing under a variety of languages, RIAs, browsers, and rendering options, I developed a coherent picture of the current options for online 2D graphics.

Implementation

The basic test consists of n transparent 2D sprites of 32 × 32 pixels each, rendered and randomly placed on a 1,280 × 960-pixel canvas. The sprites are then animated according to a simple physics system, with a point gravity at the canvas's center. The user can control the number of sprites, up to one million.

The test also measures the simulation and rendering times separately. This enables me to analyze code execution independently of graphics performance on each framework and browser. Network download time, video playback, or 3D graphics aren't tested here, just raw graphics performance for 2D sprites, as you would find in online data visualization or gaming.

The test was implemented and rendered in four frameworks (Flash, HTML5, OpenGL, and WebGL) using three programming languages (C/C++, ActionScript 3, and JavaScript). Figure 1 shows screenshots of the tests for various combinations of RIA frameworks and languages.

The Basic Simulation Loop

The simulation loop is identical in all systems. Shown in Figure 2 for C/C++, it consists of a basic Eulerian integrator and a directional gravitational force toward the canvas center, resulting in a total of seven additions, six multiplications, and one square root for each simulated particle per cycle.

Flash with Sprites

Rendering differs considerably in each framework. The simplest framework is Flash using sprites because it requires no direct rendering. Programmers can add sprites to the master `DisplayList` for each n particles, using a master bitmap, and position them dynamically at runtime (see Figure 3).

Flash with `bitmapData`

For rendering in Flash using the `bitmapData` method instead of sprites, the main program loop needs additional code to rasterize the sprites into the bitmap data object using the `copyPixels` method (see Figure 4).

HTML5

To render the particle system in HTML5, the new canvas 2D tag is used. Retrieval of the canvas and its context occurs during page initialization. The code then preloads the sprite image into a new `Image` object. The main loop uses the context's `drawImage` function to render the sprite at each particle location (see Figure 5). Unlike the Flash sprite method, and similarly to the Flash `bitmapData` method, this method doesn't reposition

 Published by the IEEE Computer Society

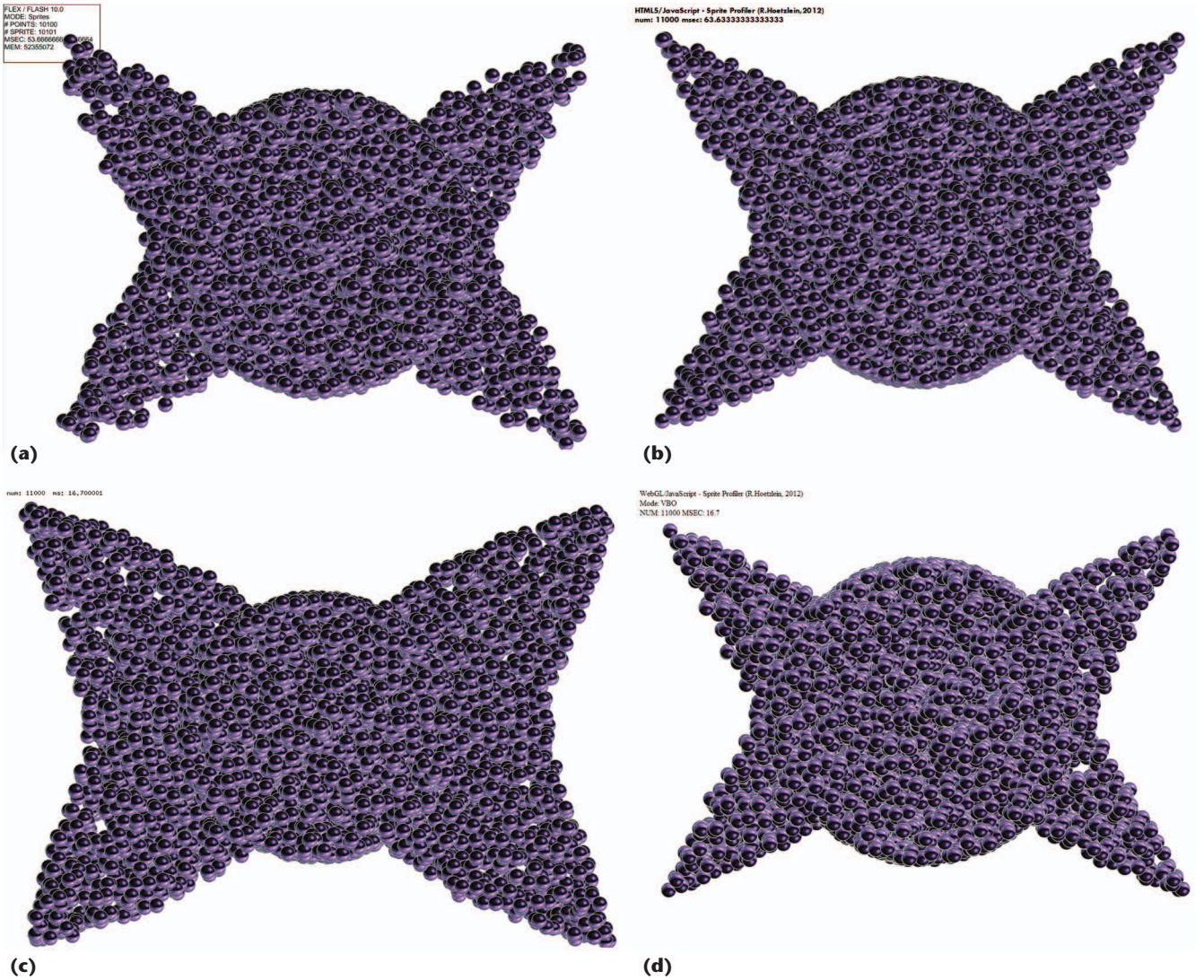

Figure 1. Rendering tests with 10,000 sprites in (a) Flash with ActionScript 3 (AS3), (b) HTML5 with JavaScript, (c) native OpenGL with C++, and (d) WebGL with JavaScript, on Google Chrome.

sprite objects and renders the bitmap directly at the particle locations per frame.

OpenGL with C/C++

The baseline test uses OpenGL in native C/C++. Because OpenGL is a low-level API, the code is somewhat more involved. A naive method sends repeated draw calls to the GPU by using `glBegin` and `glEnd` commands (see Figure 6). This requires a separate PCI bus transfer from the CPU to the GPU for each sprite. Figure 6 also shows that, because OpenGL is low level, programmers must explicitly specify each sprite's four corners and their texture coordinates.

OpenGL with Vertex Buffer Objects

A much more efficient OpenGL strategy is to use *vertex buffer objects* (VBOs), which transfer all sprite geometry as a single block of data to the GPU per frame (see Figure 7).

```
float dx, dy, dist;              // For each particle …
for (int n=0; n < num_p; n++ ) {
  dx = 640 - pos[n].x;
  dy = 480 - pos[n].y;
  dist = sqrt(dx*dx+dy*dy);
  vel[n].x += dx / dist;         // Gravitational force
  vel[n].y += dy / dist;
  pos[n].x += vel[n].x*0.1;      // Euler integration
  pos[n].y += vel[n].y*0.1;
}
```

Figure 2. The simulation loop in C/C++. This loop animates the particles using a gravity source at the screen's center.

WebGL

WebGL is similar to OpenGL but implements OpenGL ES (Embedded Systems), a simplified graphics API designed for mobile devices. Developers must specify the shaders being used to transfer individual pixels to the display. In this case, the fragment

Figure 3. The rendering setup for Flash using ActionScript 3 with sprites. The main loop, not shown here, repositions each sprite to new particle positions.

```
// Global class for embedded sprite image
[Embed(source = '../assets/ball32.png')]
private var                ballImage:Class;

// Reset rendering - Run only when N changes
while ( this.numChildren > 0 )
  this.removeChildAt ( 0 );     // Clear previous display list
this.addChild ( textFPS );      // Add frame counter to display
                                // For each particle …
for ( var n:Number=0; n < particleNum; n++ ) {
  ballSprite = new ballImage();             // Create new sprite
  ballSprite.x = particlePos[n].x;          // Position at particle
  ballSprite.y = particlePos[n].y;
  this.addChild ( ballSprite );             // Add to display list
}
```

```
// Main Loop - Run per frame
RenderBuffer.lock();
RenderBuffer.fillRect ( canvasRect, 0x222233 );
RenderBuffer.unlock();
for ( var n:Number=0; n < particleNum; n++ ) {
  RenderBuffer.copyPixels ( ballBitmap.bitmapData,
    ballBitmap.bitmapData.rect, particlePos[n] );
}
```

Figure 4. The rendering loop for Flash using ActionScript 3 with the *bitmapData* method. This loop uses the *copyPixels* method to rasterize the sprites into the bitmap data object.

```
// Setup
var canvas = document.getElementById('mycanvas');
var context = canvas.getContext( '2d' );
var ball_img = new Image();
ball_img.src = "ball32.png";

// Main loop
for( var i = 0, j = particles.length; i < j; i++ )
  context.drawImage ( ball_img, particles[i].posX,
                              particles[i].posY );
```

Figure 5. The rendering setup and main loop for HTML5 using JavaScript. (Only relevant portions of the code are shown.) This method renders the bitmap directly at the particle locations per frame.

```
glEnable ( GL_TEXTURE_2D );
glBindTexture ( GL_TEXTURE_2D, ball_glid );
for (int n=0; n < num_p; n++ ) {
  glLoadIdentity ();
  glTranslatef ( particle_pos[n].x, particle_pos[n].y, 0 );
  glBegin ( GL_QUADS );       // Transfer each quad to GPU
  glTexCoord2f ( 0, 0 ); glVertex2f ( 0, 0 );
  glTexCoord2f ( 1, 0 ); glVertex2f ( 32, 0 );
  glTexCoord2f ( 1, 1 ); glVertex2f ( 32, 32 );
  glTexCoord2f ( 0, 1 ); glVertex2f ( 0, 32 );
  glEnd ();
}
```

Figure 6. The rendering loop for OpenGL using C/C++ with naive draw methods. Because OpenGL is low level, graphic programmers must explicitly specify each sprite's four corners and their texture coordinates.

shader returns the pixel at a particular texture coordinate in the sprite.

Because WebGL doesn't contain OpenGL's direct draw methods, graphics programmers use VBOs instead. In addition, only triangles, not quads, can be rendered. So, the code must specify six corners of two triangles to draw each sprite (see Figure 8).

Discussion

Although the code becomes increasingly complex with the more low-level frameworks, GPU usage in OpenGL and WebGL clearly improves performance. Flash and HTML5 hide much graphics magic, which is their primary responsibility. In the future, these frameworks will likely become more flexible and more efficient.

Testing and Results

I measured the simulation and rendering frame rates separately at 15 data points for n sprites from 1,000 to 100,000 in each combination of framework and browser (Firefox, Chrome, and Internet Explorer 9). At times, results were outside the CPU timer's measureable limits, such as with OpenGL using VBOs, which started at 240,000 sprites. In those cases, I estimated values in the test range through linear extrapolation. I conducted the tests on a Sager NP8690 Core i7 640M laptop with a GeForce GTX 460M graphics card. The source code and results are freely available at www.rchoetzlein.com/sprites.

Scalability

Figure 9 shows some surprising outcomes regarding scalability with the number of sprites. The newly released HTML5 specification has sparked considerable debate over HTML5 versus Flash performance.[2] However, I found the browser choice to be more important to rendering performance. Firefox performed one-third as well as Chrome and Internet Explorer 9 (IE 9) when using Flash

```
// Pack 4-corners of each quad (all done on CPU)
float* dat = bufdat;
for (int n=0; n < num_p; n++ ) {
  *dat++ = pos[n].x;            *dat++ = pos[n].y;
  *dat++ = pos[n].x+32;         *dat++ = pos[n].y;
  *dat++ = pos[n].x+32;         *dat++ = pos[n].y+32;
  *dat++ = pos[n].x;            *dat++ = pos[n].y+32;
}
glEnable ( GL_TEXTURE_2D );
glBindTexture ( GL_TEXTURE_2D, img.getGLID() );
glBindBufferARB ( GL_ARRAY_BUFFER_ARB, vbo );
// Transfer all to GPU
glBufferDataARB ( GL_ARRAY_BUFFER_ARB, sizeof(float)*2*4*num_p, bufdat, GL_DYNAMIC_
                  DRAW_ARB );
glEnableClientState ( GL_VERTEX_ARRAY );
glVertexPointer ( 2, GL_FLOAT, 0, 0 );
glBindBufferARB ( GL_ARRAY_BUFFER_ARB, vbotex );
glEnableClientState ( GL_TEXTURE_COORD_ARRAY );
glTexCoordPointer(2, GL_FLOAT, 0, 0 );

glDrawArrays ( GL_QUADS, 0, num_p ); // Ask GPU to draw all sprites
```

Figure 7. The rendering loop for OpenGL using C/C++ with vertex buffer objects (VBOs). Earlier in the code (not shown), data buffers are allocated to store the 2D coordinates representing the four corners of every sprite, resulting in 2 ∗ 4 ∗ N floats. These are transferred in one block to the GPU for rendering.

```
// Main loop
for( var i=0, j=0, i < num_particles; i++ ) {
  vert[j++] = particle[i].posX;     vert[j++] = particle[i].posY;
  vert[j++] = particle[i].posX+32;  vert[j++] = particle[i].posY;
  vert[j++] = particle[i].posX+32;  vert[j++] = particle[i].posY+32;
  vert[j++] = particle[i].posX;     vert[j++] = particle[i].posY;
  vert[j++] = particle[i].posX+32;  vert[j++] = particle[i].posY+32;
  vert[j++] = particle[i].posX;     vert[j++] = particle[i].posY+32;
}
gl.bindBuffer(gl.ARRAY_BUFFER, geomVB);       // Transfer to GPU
gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.DYNAMIC_DRAW);

// Setup 2D viewport
gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
mat4.identity(pMatrix);
mat4.scale ( pMatrix, [2.0/SCREEN_WIDTH, -2.0/SCREEN_HEIGHT, 1] );
mat4.translate ( pMatrix, [-(SCREEN_WIDTH)/2.0, -(SCREEN_HEIGHT)/2.0, 0] );

// Render Vertex Buffer Object (VBO) on GPU
gl.bindBuffer(gl.ARRAY_BUFFER, geomVB);
gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
   geomVB.itemSize, gl.FLOAT, false, 0, 0);
gl.bindBuffer(gl.ARRAY_BUFFER, geomTB);
gl.vertexAttribPointer(shaderProgram.textureCoordAttribute,
   geomTB.itemSize, gl.FLOAT, false, 0, 0);

gl.activeTexture(gl.TEXTURE0); // Bind to sprite image
gl.bindTexture(gl.TEXTURE_2D, neheTexture);
gl.uniform1i(shaderProgram.samplerUniform, 0);

gl.drawArrays(gl.TRIANGLES, 0, 6*num_particles ); // Ask GPU to render
```

Figure 8. The rendering loop for WebGL using JavaScript with VBOs. Instead of transferring four corners of each sprite quad, programmers must specify six corners of two triangles to draw each sprite.

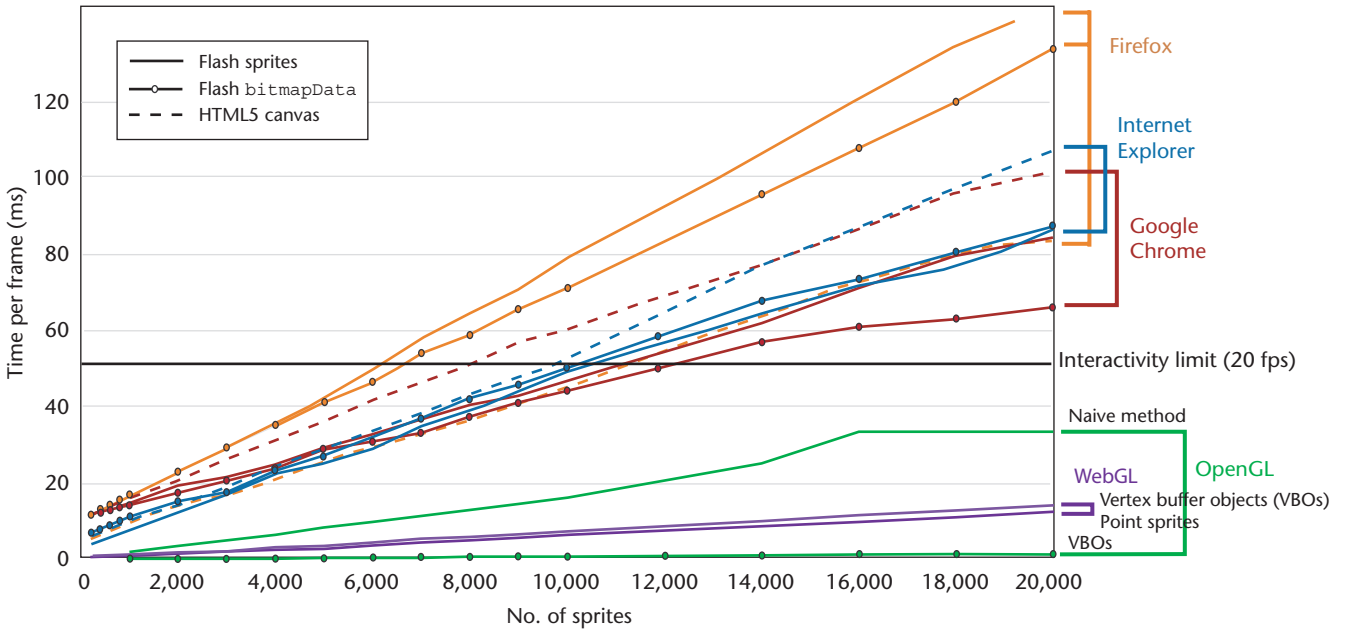

Figure 9. Graphics scalability for rich Internet application (RIA) frameworks across browsers. The time per frame is in milliseconds, so lower values are better. The number of sprites rendered ranged from 1,000 to 20,000. Overall, HTML5 and Flash performance depended heavily on the browser. Results and code samples are available at rchoetzlein.com/sprites.

with both sprites and bitmap data (see the upper orange lines). For example, with Flash, Firefox took 78 ms per frame to render 10,000 sprites, whereas Chrome and IE 9 took approximately 44 to 48 ms. However, with HTML5, Firefox took 43 ms per frame, whereas Chrome and IE 9 took 60 and 50 ms per frame, respectively (see the dashed lines). Online-game developers working in Flash have debated using sprites versus `bitmapData` (see the graph lines with tick marks in Figure 9). In my tests, the best performer was Flash with AS3 using `bitmapData` on Chrome (see the lower red line). However, in Firefox, both Flash methods were slower than HTML5. So, the combination of RIA framework, browser, and language were found to determine drawing performance; besides, developers shouldn't choose a framework on the basis of performance alone. Generally, though, `bitmapData` was approximately 10 percent faster than sprites in Flash and used considerably less memory (30 Mbytes versus 132 Mbytes for 100,000 sprites).

Of course, RIAs, which use a virtual machine and CPU rendering for OS independence, can't compare to native OpenGL with C/C++. Do you care to guess how big the difference is? As a starting point, naive OpenGL programmers often use simple draw commands that result in too many draw calls, thus overwhelming the PCI bus transfer to the GPU (see the upper green line in Figure 9). However, OpenGL professionals in the graphics industry use VBOs to package millions of polygons per frame for the graphics card. The same sprite test in OpenGL with VBOs (see the green line hugging the *x*-axis in Figure 9) rendered 10,000 sprites in 0.5 ms. The rendering time didn't exceed 50 ms (20 fps) until it reached 700,000 sprites.

WebGL debuted in 2009 to provide OpenGL-style acceleration to online applications. As expected (and as Figure 9 shows), WebGL was slower than native OpenGL because it used JavaScript for execution but was still nearly seven times faster than Flash owing to GPU acceleration. Using WebGL point sprites instead of two triangles resulted in a marginal 1-ms gain for 10,000 sprites.

Execution vs. Rendering

Examining the differences between the code execution and rendering times was found to be more instructive. One way to approach this is to ask, how much time does simulation require per frame versus rendering for 100,000 sprites? Figure 10 shows the results. The execution times were highest for RIA frameworks that ran JavaScript engines, which had to interpret code into native machine language.

Uniquely, this test methodology can provide a deductive analysis of potential problem areas and suggestions for existing browsers and frameworks. By considering a fixed frame rate of 700 ms (1.5 fps), with 100,000 sprites, I examined the relative percentage of time for each combination of browser and RIA framework (see Table 1). In this scenario, virtual code execution used between 8 to 10 percent of the total simulation time, whereas native C/C++ used less than 1 percent. Also, if one browser

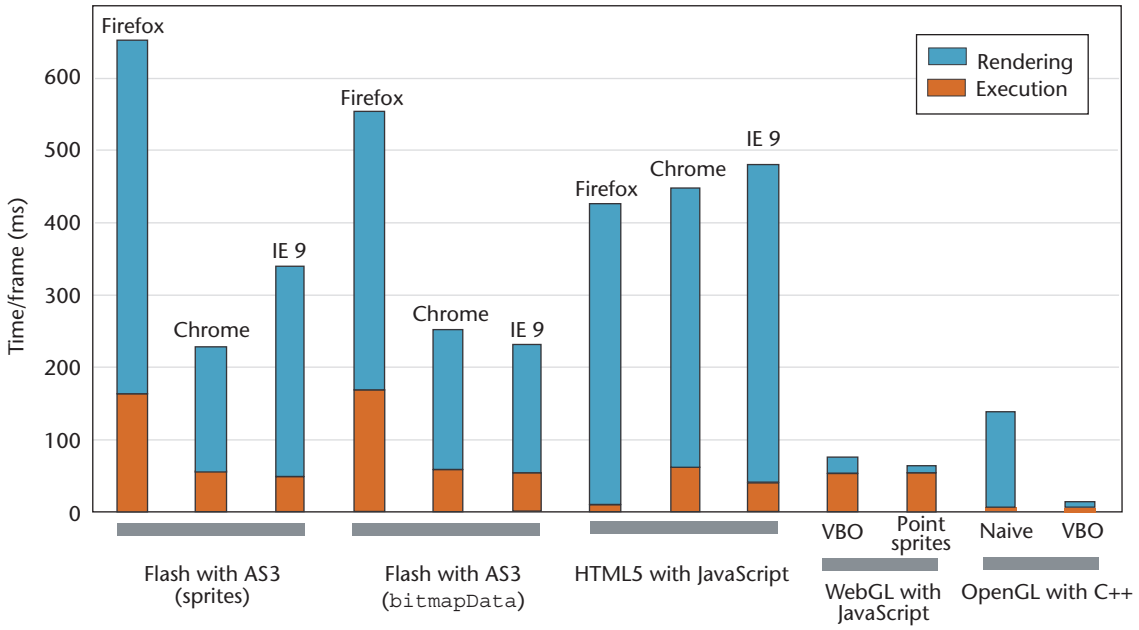

Figure 10. The simulation and rendering times per frame for various RIA frameworks and browsers for 100,000 animated sprites, 32 32 pixels each, rendered on a 1,280 900 canvas. Lower values are faster. The execution times were highest for RIA frameworks that ran JavaScript engines, which had to interpret code into native machine language.

showed dramatically higher usage relative to the others, I could deduce a potential for performance gain in that RIA framework's implementation on that browser, because the other frameworks were able to achieve the same goal. Keep in mind the same code ran on each browser. Table 1 groups the outliers.

Table 1 separately reports the rendering and JavaScript execution times. I measured rendering time by disabling simulation while enabling the rendering loop. Conversely, I measured execution time by disabling rendering. For OpenGL, there's no JavaScript engine, but I measured the rendering and simulation code loops separately. The overall performance (see Figure 9) is the sum of these rendering and code execution engines.

The most obvious outlier is Flash with Firefox, which used 70.2 percent of its time for rendering,

Table 1. Simulation and rendering times per frame for 100,000 sprites, with the percentage of time for a fixed frame rate of 700 ms (1.4 fps).

Implementation	Simulation time (ms)	Rendering time (ms)	Total time (ms)	Simulation %	Rendering %	
Flash, ActionScript 3 (AS3), sprites, Firefox	161.8	491.2	653.0	23.1	**70.2**	
Flash, AS3, sprites, Chrome	56.3	174.4	230.7	8.0	24.9	A
Flash, AS3, sprites, Internet Explorer 9 (IE 9)	48.8	291.2	340.0	7.0	41.6	
Flash, AS3, `copyPixels`, Firefox	168.0	388.0	556.0	**24.0**	55.4	
Flash, AS3, `copyPixels`, Chrome	59.5	193.5	253.0	8.5	27.6	B
Flash, AS3, `copyPixels`, IE 9	53.5	178.5	232.0	7.6	25.5	
HTML5, JavaScript, Firefox	9.7	417.0	426.7	1.4	59.6	
HTML5, JavaScript, Chrome	59.3	391.0	450.3	**8.5**	55.9	C
HTML5, JavaScript, IE 9	39.0	443.3	482.3	**5.6**	63.3	
WebGL, vertex buffer objects (VBOs), Chrome	54.0	23.0	77.0	7.7	3.3	
WebGL, sprites, Chrome	54.7	4.3	59.0	7.8	0.6	
OpenGL, naive	3.3	137.9	141.2	0.5	19.7	
OpenGL, VBOs	3.3	7.5	10.8	0.5	1.1	

A, B, and C indicate three highlighted groups of outliers, suggesting potential framework or browser improvements. Bold indicates the most obvious outliers.

compared to 24.9 percent for Chrome and 41.6 percent for IE 9 (group A in Table 1). Similarly, the Flash virtual machine running the particle system used 24.0 percent of its time for simulation on Firefox, 8.5 percent on Chrome, and 5.6 percent on IE 9 (group B). This suggests that both the execution and rendering of Flash were slower with Firefox.

With HTML5, however, Firefox achieved 1.4 percent execution time with its JavaScript engine (only 9.7 ms per frame), so the outliers here were Chrome at 8.5 percent and IE 9 at 5.6 percent (group C). Firefox is clearly optimized for JavaScript and HTML5, whereas the other two browsers are optimized for Flash. Chrome and IE 9 were head-to-head in Flash performance, except that sprite objects on IE 9 appeared to be slower than they should have been at 41.6 percent rendering time.

Download Time

This test generated a 421-Kbyte Flash .swf, whereas the HTML5-with-JavaScript files were 6.3 Kbytes total and the WebGL code was 29.4 Kbytes. As I imagined, Flash had significantly higher transfer overhead owing to the library methods that support sprites, which are capable of event handling and stylized rendering.[3] For a simple particle system, these features are unnecessary on each sprite. However, to match Flash's sprite flexibility, a typical application might need significantly more HTML5 JavaScript code.

Discussion

Clearly, trade-offs exist between flexibility, ease of use, and project management that make the decision between Flash and HTML5 more subtle. As a recent survey points out, good application development shouldn't rely on any particular RIA framework.[4]

The results suggest clear strategies for RIA developers and areas of improvement for framework toolmakers. Some conclusions from this research are that

- HTML5 rendering is still slower than Flash in Chrome and IE 9.
- Flash renders significantly more slowly in Firefox than in the other browsers.
- WebGL is the fastest online renderer owing to GPU acceleration.
- JavaScript execution time plays a significant role in online graphics (up to 10 percent of the total time per frame, depending on the application).

These tests could be extended to cover additional platforms, such as Mac and Linux, and to graphics performance on mobile devices.[5] JavaScript engine differences could also be investigated more thoroughly. The most significant gains will come when RIAs begin to fully exploit the GPU for rendering. In my experiments, for rendering 100,000 sprites over 700 ms, OpenGL used only 3 percent of that time for execution and rendering combined, whereas WebGL used 10 percent, and even the best RIA framework used 33 percent. Undoubtedly, the first RIA framework to achieve fully integrated hardware rendering will gain a significant advantage in dynamic online experiences.

People working with data visualization, such as the rendering of large social networks, should be able to render up to 10,000 nodes using `bitmapData` in AS3 with Chrome. Yet performance depends highly on the browser. To render truly large-scale networks online with tens of thousands of nodes, the only solutions are to selectively remove nodes, use view-dependent techniques, or wait for GPU rendering to mature on RIA frameworks.

References

1. J. Roettgers, "HTML5 Video Outperforms Flash on Mobile Devices," GigaOM, 14 June 2011; http://gigaom.com/video/mobile-html5-video-vs-flash.
2. S. Perez, "Does HTML5 Really Beat Flash? The Surprising Results of New Tests," ReadWriteWeb, 10 Mar. 2010; www.readwriteweb.com/archives/does_html5_really_beat_flash_surprising_results_of_new_tests.php.
3. J. Ward, "Ajax and Flex Data Loading Benchmarks," 30 Apr. 2007; www.jamesward.com/2007/04/30.
4. I. Yates, "HTML5, Flash and RIAs: 18 Industry Experts Have Their Say," 15 Oct 2011; http://active.tutsplus.com/articles/roundups/html5-and-flash-17-industry-experts-have-their-say.
5. S. Christmann, "GUIMark 3—Mobile Showdown," Craftymind, 2011; www.craftymind.com/guimark3.

Rama C. Hoetzlein *is a project scientist with the Department of English Transliteracies project at the University of California at Santa Barbara. He performed this research while in the Department of Architecture and Media Technology at Aalborg University Copenhagen. Contact him at rama@rchoetzlein.com.*