SceneJS: A WebGL-Based Scene Graph Engine

40

Lindsay Kay

40.1 Introduction

The WebGL graphics API specification extends the capabilities of the JavaScript language to enable compatible browsers to generate 3D graphics on the GPU without the need for plugins. With JavaScript execution speed a potential bottleneck, high-performance WebGL applications rely on executing minimal JavaScript while offloading as much work as possible to the GPU in the form of shader programs written in GLSL.

This chapter describes key concepts of SceneJS, an opensource 3D engine for JavaScript that applies some simple scene-graph concepts such as state inheritance on top of WebGL [Kay 10]. The framework focuses on efficient rendering of large numbers of individually pickable and articulated objects as required for high-detail, model-viewing applications such as BIMSurfer and the BioDigital Human shown in Figure 40.1 [Berlo and Lindeque 11, BioDigital 11].

Essentially, SceneJS works by maintaining a state-optimized list of WebGL calls that is updated through a simple scene graph API based on JSON [Crockford 06]. As updates are made to the graph, SceneJS dynamically rebuilds only the affected portions of the call list, while automatically taking care of shader generation.

This chapter describes the general architecture of SceneJS, focusing mainly on the JavaScript strategies it uses to efficiently bridge its abstract scene representation with an efficient use of WebGL and how those strategies can be exploited through its API.

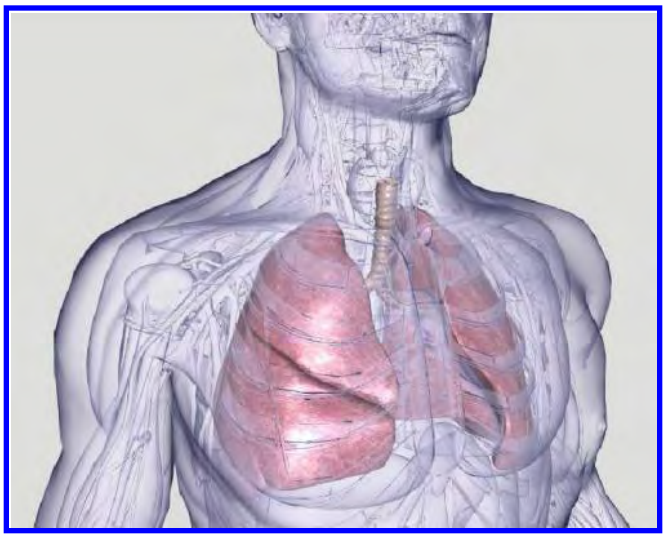

Figure 40.1. SceneJS is the rendering engine within the BioDigital Human, a free Web-based atlas of human anatomies and conditions. When all visible, the 1,886 meshes and 126 textures in the Human's male anatomy view render at around 10–15 FPS in Chrome 14.0.835.202 on a computer with an i7 CPU and an NVIDIA GeForce GTX 260M GPU.

40.2 Efficiently Abstracting WebGL

A scene graph is a data structure that arranges the logical and spatial representation of a graphical scene as a collection of nodes in a graph, typically a tree. A key feature of most scene graphs is *state inheritance*, in which child nodes inherit the states set up by parents (e.g., coordinate spaces, appearance attributes, etc.). Scene graphs typically provide a convenient abstraction on top of low-level graphics APIs, which encapsulates optimizations and API best practices, leaving the developer free to concentrate on scene content.

WebGL is based on OpenGL ES 2.0, which offloads most of the rendering work to the GPU in the form of shaders written by the graphics programmer. Thus, WebGL is geared for the limited execution speed of JavaScript, encouraging JavaScript in the application's graphics layer to be used for little more than directing GPU state: buffer allocation and binding, writing variables, draw calls, and so on.

SceneJS bridges the gap between its scene-graph API and WebGL through a five-stage pipeline:

1. Scene definition. A JSON definition like that of Listing 40.1 is parsed to create a scene graph like Figure 40.2 with resources such as vertex buffer objects (VBOs) and textures stored for its nodes on the GPU. Note the geometry nodes at the leaves.

```
SceneJS.createScene({ // Scene graph root
  type : "scene",
  id : "the-scene",
  canvasId : "my-canvas", // Bind to HTML5 canvas
  nodes: [{ // View transform with node ID
      type: "lookAt",
      id: "the-lookat",
      eye : { x : 0.0, y : 10.0, z : 15 },
      look : { y : 1.0 },
      up :  { y : 1.0 },

      nodes: [{ // Projection transform
          type : "camera",
          id : "the-camera",
          optics : {
            type: "perspective",
            fovy : 25.0,
            aspect : 1.47,
            near : 0.10,
            far : 300.0
          },

         nodes: [{ // Light source
             type : "light",
             mode : "dir",
             color : { r : 1.0, g : 1.0, b : 1.0 },
             dir : { x : 1.0, y : -0.5, z : -1.0 }
           },

           { // Teapot geometry
              type : "teapot"
           },

           { // Texture for two cubes
             type : "texture",
             uri : "images/texture.jpg",

             nodes : [{ // Translate first cube
                 type : "translate",
                 x : 3.0,

                 nodes : [{ // Cube geometry
                   type : "cube"
                 }]
               },

               { // Translate second cube
                 type : "translate",
                 x : 6.0,

                 nodes : [{ // Cube geometry
                   type : "cube"
                 }]
               }]
           }]
       }]
   }]
});
```

Listing 40.1. Scene-graph definition. The scene graph is a DAG expressed in JSON, in this case defining a scene containing one teapot and two textured cubes, all illuminated by a directional light source and viewed in perspective. Geometry nodes are normally at the leaves, where they inherit the state defined by higher nodes.

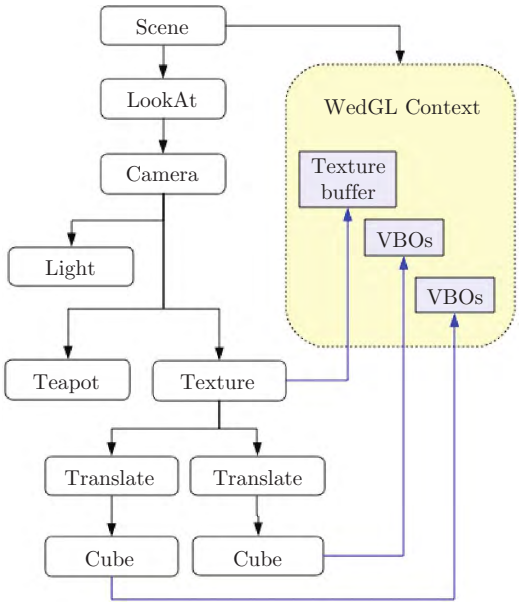

Figure 40.2. Scene graph compiled from the scene definition of Listing 40.1. Note that the geometries at the leaves inherit state from parent nodes and that various nodes hold resources allocated for them on the GPU.

2. **Draw list compilation.** The scene graph is traversed to compile a sequence of WebGL state changes. This is described in more detail in Section 40.2.1.

3. **Call list compilation.** The draw list is compiled into a fast list of WebGL calls with arguments prepared from the draw list states. As shown in Listing 40.2, call list nodes are functions that wrap WebGL calls and are created by higher-order functions that prepare and memorize their arguments in closures.

4. **State sorting.** The call-list nodes are sorted on their corresponding draw list states to minimize the number of state changes that will go down the OpenGL pipeline, as described in Section 40.2.2.

5. **Call list execution.** The call list is executed to render the frame.

Once the scene is created, we start its render loop, which executes these stages to render the first frame. Note that each stage caches its results. Then, with the render loop running, we can receive scene-state updates through the API's scene accessor methods, as shown in Listing 40.3, which we buffer for batch processing at the start of each loop.

```
//..
callList.push(
  (function() {
    // Call arguments prepared and cached in a closure.
    var uEyePosLoc = currentDrawListNode.shader.getUniformLocation("uEyePos");
    var eye = currentDrawListNode.lookAt.eye;

    // The WebGL call. The eye is a state object that is shared
    // by reference with the scene graph's lookAt node and the
    // draw list node.
    return function() {
      glContext.uniform3fv(uEyePosLoc, [eye.x, eye.y, eye.z]);
    }
})());
```

Listing 40.2. Call-list compilation. Each WebGL call is wrapped by a function that is created by a higher-order function that prepares the arguments and caches them in a closure. In this example, for efficiency, the higher-order function finds and caches the location of a shader uniform and gets the lookAt state's eye position in a variable for faster access.

Different types of scene-state updates require re-execution from different stages of the pipeline in order to synchronize the rendered view. As we process the buffered updates, we minimize JavaScript execution by re-executing the pipeline from the latest stage, which will synchronize the view for all updates. Note that the pipeline is not executed when the buffer is empty.

Most types of scene update are written straight through to the draw and call lists without requiring retraversals to rebuild states and without the addition/removal of

```
// Find the scene and start it:
var scene = SceneJS.scene("the-scene");
scene.start();

// Find the lookAt node and get its eye attribute:
var lookAt = scene.findNode("the-lookat");
var eye = lookAt.get("eye");

// Set the lookAt's eye attribute:
lookAt.set({ eye: { x : eye.x + 5.0 } });

// Create another light:
scene.findNode("the-camera").add({
  nodes: [{
    type : "light", dir : { y : -1.0 }
  }]
});
```

Listing 40.3. Scene-graph accessors. The scene graph is encapsulated by API functions that provide read and write access to node states, render loop control, picking, and so forth. Note that write access is not provided for states that would require reallocation of GPU-bound resources.

list nodes. This is supported by sharing state between the lists via common objects, restrictions on what state may be updated, and the simple approach to state inheritance described in Section 40.2.1. Therefore, for most types of update, including camera movements, color changes, and object visibility,[1] we need only re-execute the pipeline from Stage 5.

This approach is tuned to applications in which the scene-graph structure does not change often, where it suffices to hold content in the graph most of the time, toggling its visibility to enable or disable it.

The most expensive type of update involves the addition of nodes to the scene. For this type, we must re-execute the pipeline from Stage 1 to reparse the JSON definitions for the new nodes, then rebuild the draw and call lists. Almost as expensive are node relocations and removals, for which we need to re-execute from Stage 2.

SceneJS is a lean rendering kernel that does not include visibility culling and physics. However, the efficiency with which it processes updates to object visibilities and transforms makes it practical for integrating external libraries for these tasks, such as jsBVH for culling [Rivera 10] and ammo.js for physics [Zakai 11].

40.2.1 Draw-List Compilation

We construct the draw list by traversing the scene graph in depth-first order while maintaining a stack for each scene-node type, pushing each node's state to the appropriate stack on pre-visit, then popping it again on post-visit. At each geometry, we create a draw-list node that references the state at the top of each stack. We also generate a GLSL shader for the draw-list node, tailored to render the configuration of states that the node references. For reuse by other nodes with similar states, we hash the shader on those states and store it in a pool.

Each draw-list node has everything needed for a draw call to render a scene object. Figure 40.3 shows the state-sorted draw list compiled from the scene graph of Figure 40.2.

For most node types, the state at the top of the stack completely overrides the states lower in the stack, resulting in geometries inheriting state only from the closest parent of that type. As mentioned in Section 40.2, this means that updates to inherited states of these types will write straight through to the draw and call lists via shared state objects, requiring neither scene retraversal nor addition/removal of draw- or call-list nodes.

Two node types are special cases, however:

1. For modeling transform nodes, we maintain a stack of matrices; at each of these nodes, we multiply that node's matrix by the top of the stack before pushing it. The top matrix at each leaf geometry is then referenced by a draw-list node. An update to any transform node (e.g., changing a rotation angle),

[1] The call list nodes are actually indexed by their draw-list nodes: as we execute the calls, we can therefore efficiently skip the calls associated with invisible draw-list nodes.

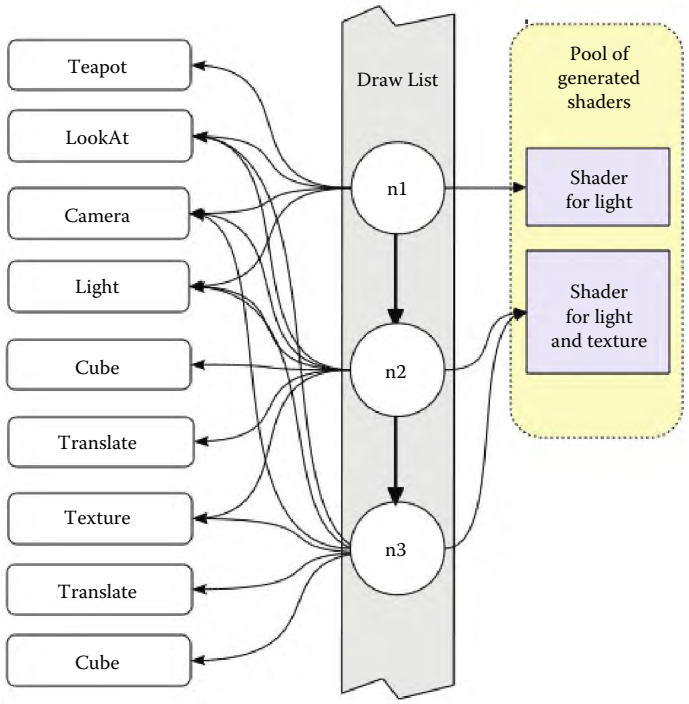

Figure 40.3. State-sorted draw list compiled from the scene graph of Figure 40.2. Nodes `n1`, `n2`, and `n3` reference the states needing to be set on WebGL to draw the teapot and cubes at the leaves of the scene graph. Nodes `n2` and `n3` reference a similar state configuration and therefore reference the same shader.

therefore, requires us to re-execute the SceneJS pipeline from Stage 2 to retraverse the branch to recalulate stale draw-list matrices.

2. Geometry nodes may be nested to support VBO sharing, as described in Section 40.3.2, which is where a parent geometry node defines vertex arrays that are inherited by child geometries. For this case, draw-list nodes are created for leaf geometries as usual, except that as we stack geometry states, we accumulate on those the arrays belonging to any state already on the top of the stack. Updates to the vertex arrays on the geometry nodes are still efficient, however, since the arrays themselves are shared by reference amongst the scene, draw- and call-list nodes.

40.2.2 State Sorting

State sorting involves minimizing the number of state changes that go down the OpenGL pipeline by grouping similar states within the call list. We sort the call-list nodes by shader, texture, then VBO. Shader is our primary order because switching shaders causes widespread disruption of the OpenGL pipeline, necessitating the rebind of all other states. We order on textures next because during development, we observed that they were slower to bind than VBOs.[2] A further state sort is performed when we execute the call list, in which we track the ID of the last state change that we made on WebGL so that we don't make the same change twice.

40.3 Optimizing the Scene

The API supports several scene-definition techniques that improve scene performance by exploiting the state sorting order and the pipeline described in Section 40.2.

40.3.1 Texture Atlases

A texture atlas is a large image that contains many subimages, each of which is used as a texture for a different geometry, or different parts of the same geometry. The subtextures are applied by mapping the geometries' texture coordinates to different regions of the atlas. As mentioned earlier, SceneJS sorts the draw list by shader, then by texture. As long as each of the geometry nodes inherits the same configuration of parent-node states and can therefore share the same shader, the draw list will bind the texture once for all the geometries. Another important benefit of texture atlases is that they reduce the number of HTTP requests for texture images [NVIDIA 04].

40.3.2 VBO Sharing

VBO sharing is a technique in which a parent geometry node defines vertices (consisting of arrays of positions, normal vectors, and UV coordinates) that are inherited by child geometry nodes, which supply their own index arrays pointing into different portions of the vertices. The parent VBOs are then bound once across the draw calls for all the children. Each child is a separate object, around which, as shown in Listing 40.4, each child geometry can be wrapped by a different texture or material, etc. This is efficient to render as long as each child geometry inherits a similar combination of states and thus avoids needing to switch generated shaders, as described in Section 40.2.1.

[2] In Chrome 14.0.835.202, running on Ubuntu 10.0.4 with NVIDIA GeForce GTX 260M GPU.

```
{
  type : "geometry",
  positions : [...], // All positions
  normals : [...], // All normals
  uv : [...], // All UVs

  nodes: [{
       type : "texture",
       uri :  "someTexture.jpg",
       nodes : [{
         type : "geometry",
         primitive : "triangles",
         indices : [...] // Faces for this geometry
       }]
     },
     {
       type : "texture",
       uri : "anotherTexture.jpg",
       nodes : [{
         type : "geometry",
         primitive : "triangles",
         indices : [...] // Faces for this geometry
       }]
     },
     {
       type : "texture",
       uri :  "oneMoreTexture.jpg",
       nodes : [{
         type : "geometry",
         primitive : "triangles",
         indices : [...] // Faces for this geometry
       }]
     }
  ]
}
```

Listing 40.4. VBO sharing to reduce binding calls, as described in Section 40.3.2. In this example, a parent defines VBOs of positions, UVs, and normals that are inherited by the children, which define their primitives through index arrays pointing into different portions of the VBOs. Each child also applies a different texture to its portion.

40.3.3 Sharable Node Cores

Traditionally, reuse within a scene graph is done by attaching nodes to multiple parents. For dynamically updated scenes, this can have a performance impact when the engine must traverse multiple parent paths in the scene graph, so SceneJS takes an alternative approach with *node cores*, a concept borrowed from OpenSG [OpenSG 10].

A node core is the node's state. Having multiple nodes share a core means that they share the same state. This can have two performance benefits:

1. An update to a shared node can write through to multiple draw- and call-list nodes simultaneously.

```
// Define a couple of nodes, in a library
// to prevent them rendering by themselves:
{
  type : "library",
  nodes : [
    {
      type : "geometry",
      coreId  : "my-geometry-core",
      positions : [..],
      indices : [..],
      primitive : "triangles"
    },
    {
      type : "material",
      coreId : "my-material-core",
      baseColor : { r: 1.0 }
    }
  ]
},

// Share their cores:
{
  type : "material",
  id : "my-material",
  coreId : "my-material-core",
  nodes : [
    {
      type : "geometry",
      coreId : "my-geometry-core"
    }
  ]
}
```

Listing 40.5. State reuse through shared node cores, described in Section 40.3.3. We define `geometry` and `material` nodes within a `library` node, which prevents them from being rendered. The `geometry` and `material` each have a `coreId` which allows their state (VBOs, color etc.) to be shared by other nodes of the same type later in the scene.

2. There is increased chance of identical repeated states having matching IDs when executing the call list, which, as described in Section 40.2.2, tracks the state IDs to avoid redundantly reapplying them.

Listing 40.5 shows an example of node-core sharing through the scene-definition API.

40.4 Picking

We use a variation of the pipeline described in Section 40.2 for mouse picking. When a pick is made, we compile the scene graph to special *pick-mode* draw and call lists, which render each pickable object in a different color to an offscreen pickbuffer. We then read the pixel at the pick coordinates and map its color back to the picked object.

Frameworks using this technique typically reduce the viewport to a 1×1 region at the pick coordinates for efficiency while rendering the pickbuffer. SceneJS uses the entire original viewport so that it can retain the pick buffer to support fast repicking at different coordinates for the case when nothing has changed in the image since the last pick. This supports fast mouseover effects such as tool-tips.

40.5 Conclusion

When aiming for high-performance 3D graphics in the browser, the greatest performance bottleneck is JavaScript overhead. To overcome this bottleneck, WebGL applications can greatly benefit from clever caching strategies and the use of optimizations like compilation to closures. The retained mode API of SceneJS benefits from this kind of preprocess optimization where complex dynamic code can be compiled to a fast static form. That said, classical techniques such as vertex sharing and texture mapping still have the same impact as in any other OpenGL application and should still be applied.

SceneJS is opensource software with many more features than were described here. Moving forward, its road map will continue to focus on high-detail, model-viewing applications; extend its optimizations for scenes in which nodes are frequently added, relocated, and removed; and leverage emerging technologies such as Web Workers and Google Native Client for additional performance.

Bibliography

[Berlo and Lindeque 11] Leon Van Berlo and Rehno Lindeque. "BIMSurfer." http://bimsurfer.org, September 8, 2011.

[BioDigital 11] BioDigital. "BioDigital Human." http://biodigitalhuman.com, August 31, 2011.

[Crockford 06] Douglas Crockford. "RFC 4627." http://tools.ietf.org/html/rfc4627, July 2006.

[Kay 10] Lindsay Kay. "SceneJS." http://scenejs.org, January 22, 2010.

[NVIDIA 04] NVIDIA. "Improve Batching Using Texture Atlases." ftp://download.nvidia.com/developer/NVTextureSuite/Atlas_Tools/Texture_Atlas_Whitepaper.pdf, September 7, 2004.

[OpenSG 10] OpenSG. "OpenSG: Node Cores." http://www.opensg.org/htdocs/doc-1.8/NodeCores.html, February 8, 2010.

[Rivera 10] Jon-Carlos Rivera. "jsBVH." https://github.com/imbcmdth/jsBVH, April 4, 2010.

[Zakai 11] Alon Zakai. "ammo.js." https://github.com/kripken/ammo.js, May 29, 2011.