Native browser support for 3D rendering and physics using WebGL, HTML5 and Javascript

Rovshen Nazarov
American University in Bulgaria
Blagoevgrad 2700
Bulgaria

ran090@aubg.bg

John Galletly
American University in Bulgaria
Blagoevgrad 2700
Bulgaria
+359 73 888 466
jgalletly@aubg.bg

ABSTRACT

In the last few years, JavaScript libraries have been developed to enable developers to create and manipulate 3D objects in the browser. These JavaScript libraries incorporate physics and 3D processing algorithms, HTML 5 elements and technologies (such as canvas and background workers), and the Web Graphics Library (WebGL). This paper provides an insight into these technologies, and describes the experience gained in using the latest innovations for client-side programming - using the browser as an application processing unit to enable plugin-free running of a high-quality and computing-intensive game application with 3D graphics rendering and physics effects.

Categories and Subject Descriptors

General Terms

Performance, Design, Human Factors, Languages.

Keywords

Client-side programming, 3D objects with physics effects, JavaScript libraries, HTML 5, Web Graphics Library (WebGL), rendering engines, physics engines

1. INTRODUCTION

Initially, web browsers were not intended for 3D graphics applications, but were designed for rendering simple web pages with static content. With the advent of dynamic content and client-side scripting languages, the demand for some sort of 3D graphics support in browsers started growing steadily.

One of the first technologies for rendering 3D in browsers was VRML, which has now been superseded by X3D. Other popular technologies that were introduced after VRML are ShockWave, Flash, Silverlight, QuickTime and others. Additionally, 3D objects may be developed using programs such as Maya, 3ds Max or Blender, and then imported into a browser. There are also several 3D game engine plugins, such as Unity3D, available for the browser.

All of the above technologies require browser plugins to play. Unfortunately, there is no standard that sets common practices for the development of 3D applications using plugins. There are many different plugins for 3D rendering, and therefore a user will need to install a different plugin to use a certain 3D application in the browser. Different plugins introduce compatibility issues and make using and developing such applications difficult. Even more, plugins can be a security problem. So, instead of relying on plugins, developers began to look towards JavaScript as a means to create and manipulate 3D objects directly in the browser.

One JavaScript 3D rendering package that has been developed in recent years is WebGL (Web Graphics Library) [1]. It is derived from OpenGL ES 2.0 and provides similar rendering functionality but specifically for web applications. It uses the HTML5 Canvas element for rendering 3D elements, and, importantly, it has a JavaScript API, thus allowing JavaScript programs running in a browser to gain access to the GPU without a plugin. With WebGL there is no need to install a plugin - 3D applications can be run directly from the browser. WebGL has gained a huge interest in the development community and there are already many stunning 3D demonstrations available on Internet.

2. WebGL

At the core of the WebGL technology are scripts known as shaders - they define how everything is drawn on the screen. There are two shaders in WebGL: the vertex and fragment shader These shaders are responsible for position calculation and colour specification respectively. The vertex shader basically converts the points in the 3D model into 2D screen coordinates. The fragment shader tells WebGL what colour a given point in the model should be. (Figure 1).

As noted above, WebGL is based on OpenGL, which is a rather old type C-style library. It features a long list of functions used to set different states and pass data to the GPU. WebGL is not 3D in itself, but it is a drawing API that gives JavaScript access to the hardware accelerated graphics GPU. Thus, a developer cannot just go ahead and use WebGL to create a 3D model without a strong knowledge of geometry and C-like language.

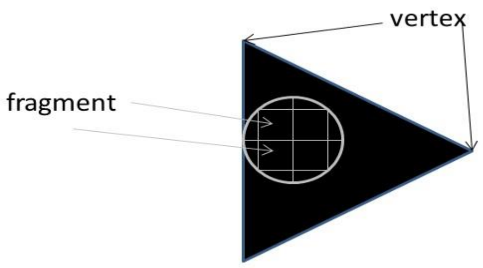

Figure 1 Shader Structure

Shaders do not get executed in the browser like JavaScript, but they need to be precompiled using JavaScript code and shader functions and attributes from WebGL. After their initial pre-compilation, the shaders need to be linked. Attributes and uniforms, which are part of the WebGL technology, are the way to bridge JavaScript run in the browser and the shaders run on the GPU.

However, developing in pure WebGL is not an easy task as it is based on the vertex and fragment shader scripts, requiring the developer to be familiar with the mathematics of 3D geometry. Luckily there is solution since many developers have dedicated their free time to create open-source libraries written in JavaScript, built on top of WebGL (Figure 2).

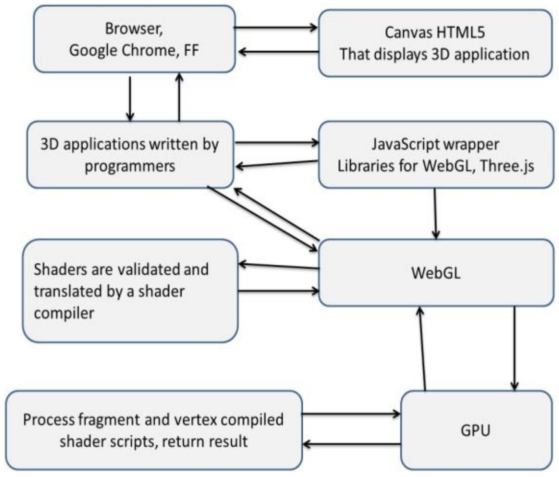
Browser, Google Chrome, FF
Canvas HTML5 That displays 3D application
3D applications written by programmers
JavaScript wrapper Libraries for WebGL, Three.js
Shaders are validated and translated by a shader compiler
WebGL
Process fragment and vertex compiled shader scripts, return result
GPU

Figure 2 WebGL Stack

3. WebGL JAVASCRIPT LIBRARIES

3.1 Rendering Engines

Developing in pure WebGL can be time consuming, and therefore, to make it accessible to a wider range of developers, many wrapper libraries that turn WebGL into an object oriented-like library have been created.

One of the most popular, most developed and supported of the WebGL wrapper JavaScript libraries is Three.js[2]. Three.js allows the creation of 3D objects and effects with a few lines of code. Using Three.js plugins for 3D object development software such as Maya and Blender, makes it possible to convert a 3D object created on those platforms into a JSON object and then load that object using the Three.js JSON loader into a web application.

CubicVR.js[3] is the main competitor for the Three.js rendering engine. It supports both rendering and physics (using Ammo.js for physics). However, its documentation is limited to a basic description of the class parameters. CubicVR.js is a port from the CubicVR 3D Engine, C++ Version.

Scene.js[4] is an open-source 3D engine for JavaScript which provides a JSON-based scene graph API on WebGL. Scene.js specializes in efficient rendering of large numbers of objects as required by high-detail model-viewing applications in engineering and medicine. Game engine effects like shadows, reflections etc. are not supported. Using a JSON API makes it easier to integrate it well with AJAX, parsers and databases.

CopperCube[5] is a proprietary package that allows the creation of 3D applications without programming. It is aimed primarily at non-programmers, for creating 3D applications in a drag-and-drop fashion based on a behavior and action-based logic system.

3.2 Physics Engines

The ability to draw 3D graphics is not enough for realistic-looking dynamic applications. The application may also need to simulate the effects of physics – the effects of gravity and collisions, for example.

A very popular JavaScript physics library for WebGL is Ammo.js[6], which is a direct port of the famous Bullet physics engine[7] to JavaScript. Ammo.js is used almost always in conjunction with Three.js. However, Ammo.js is a non-trivial physics library and requires some knowledge of the Bullet physics library class structure. Furthermore, it is not well-documented for use with the 3D rendering libraries such as Three.js. You have to have certain skills to benefit from most of the capabilities offered by Ammo.js.

PhysiJS.js[8] is an alternative physics framework for Three.js, having a much simpler user interface than Ammo.js. In fact, PhysiJS.js is built on top of Ammo.js but runs the physics simulation in a separate thread (via a JavaScript Web Worker) to avoid impacting on the application's performance. Again, a lack of documentation means that the developer has to look at the code of the library and understand its logic. In addition, if PhysiJS.js is used, usage is limited to only those objects and functions defined in PhysJS.js.

One more physics engine, JigLibJS2.js[9], is claimed to be faster than Ammo.js, but speed comes with a limitation on the functionality of the library. JigLibJS2.js is a JavaScript 3D physics engine port of JiglibFlash. The code is generated automatically from the AS3 code so all functionality works exactly like the AS3 version.

A recent entry to JavaScript physics engine libraries is Cannon.js[10]. It is described as a lightweight and simple 3D physics engine, and smaller in size than ported physics engines such as Ammo.js and JigLibJS, but it is still under development and, in the future, it may become full featured library.

In August 2012, Prall [11] (who actually developed PhysJS.js) wrote an interesting comparison of four physics engines: Box2dweb, Ammo.js, JigLibJS, and Cannon.js. (The first of which only simulates two-dimensional scenes, and the last one was still in development at the time of the comparison.) Prall's conclusion stated "Box2dweb does not support 3D worlds; Ammo.js can suffer from performance issues, and JigLibJS.js is affected by its API design and lack of functionality. Challengers such as Cannon.js and others which are 100% written in JavaScript are beginning to appear, but none are yet ready to be widely used".

Both Ammo.js and JigLibJS2.js are ports of existing physics engines. Because they are ports, and not hand-coded, they are not optimized for the web. Of the above, PhysiJS.js and Cannon.js are directly written in JavaScript.

4. EXPERIENCES

In order to get real experience of some of the new technologies related to WebGL discussed in this paper, a basic racing car game with physics effects was implemented. Unsurprisingly perhaps, many problems were encountered, especially at the beginning of the implementation in 2011.

The main problem was the lack of documentation for the WebGL-based JavaScript libraries, and also for the few demonstrations that were available at that time. Often, it was necessary to look at the developer's code to understand how the WebGL wrapper libraries worked. Fortunately, the WebGL developers' community is very supportive and united, and the main library developers would answer questions and offer constant feedback very quickly. But even now, while developing a game application with 3D graphics rendering and physics effects that uses Three.js and Ammo.js, a developer is still forced to look into the library code to understand how a more complex scene could be created. For the racing car game referred to above, it was necessary to look at three different implementations of car 3D objects that were implemented using the Three.js library to understand how to create a connection between the Ammo.js and Three.js libraries.

When it came to the physics, things became more complex. Ammo.js is port from a C++ library, with very limited documentation. Thus, to use this physics library with the rendering library (Three.js), it was necessary to experiment with different settings and configurations for creating and tracking two objects: a 3D object that the user sees on the screen, and a separate physics object that manages the effects such as collisions. Moreover, these two objects have to be like one, so it was important to ensure that the two different libraries were well integrated to achieve even a simple collision effect.

On the other hand, creating simple 3D scenes with basic collision (using Ammo.js and Three.js) has become easier today than it was in 2011. A basic scene with a ground of any material and with some lighting can be easily created with reference to the numerous short tutorials available for Three.js on Internet. Creating a 3D object, such as a box, is as simple as a few lines of code.

An object's skeleton is created using Three.js library's ShapeNameGeometry constructor with the appropriate shape parameters, such as radius for a sphere. Then the MeshLambertMaterial constructor is used to load an image and convert it to a Three.js object. This material object is wrapped around the 3D skeleton like a skin using the library's Mesh constructor. This mesh object represents the final product, a 3D object that may be added to the 3D scene at a certain position.

Even though creating a 3D scene is trivial, adding physics to it using Ammo.js is non-trivial and requires some knowledge about both libraries and how they can be synchronized. There are very few tutorials explaining examples of the above mentioned combination. Thus, some analytical skills and good JavaScript knowledge is needed to look into the code of the demonstrations, which implement both the physics and rendering to understand on what basis the two libraries can collaborate.

To add physics using Ammo.js, one first needs to set up a physics world (as opposed to the Three.js 3D scene world created above), as a separate process. To do this, several Ammo.js's objects need to be created, such as the collision configuration; a dispatcher; an overlapping cache pair; a sequential impulse constraint solver and the most important discrete dynamics world (which is a physics scene of an infinite size). After setting up the physics world, adding objects is easy and comes down to several lines of code. However, adding just physics objects is not enough and the developer needs to make sure that the physics objects are synchronized with the 3D objects. The easiest way to do this is to copy the values of the objects' x, y, z positions and quaternion (which is also represented by x, y, z and an angle) from the physics object to the 3D object. The important point is to do this during the animation phase for both 3D and physics objects because the physics engine calculates all effects and the 3D objects should reflect those calculations.

If one decides on adding a more complex customized 3D object, one can use, for example, a free 3D model from Internet and convert it using Blender to Three.js's JSON format. After that the object can be loaded using another Three.js library extension called JSON loader. This loader will generate geometry and material Three.js objects that can be used later to create a Three.js mesh, which is a 3D object.

In addition to simple objects, Ammo.js also allows generation of complex classes, such as car classes implemented in the racing game mentioned above. The idea behind the Ammo.js vehicle class is very simple - one provides all car-related parameters, such as engine power, suspension stiffness, and other similar real car-like parameters and Ammo.js controls the car interaction with other objects in the scene. The trickiest part is to handle the synchronization of Three.js library generated 3D objects and Ammo.js shapes that represent wheels. However, by looking at the Bullet physics documentation, or by asking on the Three.js forum, one can find a solution for any obstacle.

Moreover, user interaction, the most interesting aspect of 3D or any animation, can be easily implemented using Three.js's First Person Control library, or a JQuery's mouse control extension, or one's own implementation of key event handling procedures. Once one sets up the car properly, one can also define how the physics objects should behave on collision with respect to their velocity and mass. For instance, the objects with zero mass are known to be static and are not expected to be moved, and therefore, if an object hits such static object, it should bounce off with different velocity than it would from a non-static objet.

Once the physics and 3D scene is set and ready, one can add extra features to the scene to make the scene feel more realistic. Three.js allows adding different filters, postprocessors and shaders for enhancing the quality of generated 3D objects. An interesting addition to a 3D object would be reflection. In addition, one may want to add audio to the scene, or to the car engine when the forward button is pressed. Adding these extra features is non-trivial, but Three.js has many demonstrations that implement all of the effects that one might want to use. And again, one will have to look into the code to see how the effects are applied and try to use that knowledge to enhance one's own scene. In the car racing game, a sky effect, shades, panoramic surround, and object reflection have been added to make the scene feel more realistic.

A key factor for making WebGL, and support for it grow are the enthusiasts who spend their free time developing demonstrations and writing WebGL wrapper libraries.

5. THE FUTURE

Although WebGL is a big step forward in terms of 3D rendering for browsers, it is not the end of the story. One thorny problem is that of security. Arguments rage about the vulnerability of WebGL in terms of security issues. Microsoft has taken the position that WebGL is too great a security risk and the existing defences are not robust enough – Internet Explorer does not support WebGL. On the other hand, Mozilla takes the position that the defences they have put in place will be adequate, and that WebGL provides is an important tool for the web.

Another drawback is performance. Execution of WebGL applications can overload low-performance machines, and most of typical video cards cannot support the full range of 3D effects offered by WebGL.

Moreover, adding physics and many 3D objects may slow down performance significantly. In some cases, after running a 3D scene for few minutes, a browser may start freezing and be unresponsive or slow to respond. This is a huge problem for complex 3D games, and therefore, even more needs to be done to make browsers run JavaScript even faster and more efficient.

There is still a lot to improve in terms of browser-level hardware access and the 3D rendering capabilities of WebGL. Many developers today expect web browsers to provide all the features necessary for creating a desktop level-like 3D experience in the browser. To fulfill this expectation, much more needs to be done.

Moreover, sufficient documentation needs be developed so that more complex 3D applications may be created by less-experienced developers, which is not the case for the moment. However, many initiatives have been started by web development enthusiasts.

6. CONCLUSION

In the last few years, much has improved in terms of JavaScript rendering and physics engines, but there is still some way to go. Although very complex and dynamic 3D scenes may not yet be achievable, a number of interesting developments are now presenting themselves. The blending of easily accessible technologies in today's browsers makes a very interesting environment for realistic 3D modeling and viewing. Moreover, continued support from the browser development companies, Google and Mozilla, for WebGL, and the continued improvement and extension of hardware access from the browser is enabling developers to create the first prototypes of desktop-level, high-quality 3D applications with physics for the browser. As pointed out at the beginning, the main advantage of this technology is that there is no need to install or download anything for viewing a high-quality 3D application.

7. REFERENCES

[1] WebGL home page
http://www.khronos.org/webgl/

[2] Three.js home page
http://mrdoob.github.com/three.js/

[3] CubicVR.js home page
http://www.cubicvr.org/

[4] Scene.js home page
http://scenejs.org/

[5] CopperCube home page
http://www.ambiera.com/coppercube/

[6] Ammo.js home page
https://github.com/kripken/ammo.js/

[7] Bullet physics engine home page
http://bulletphysics.org/wordpress/

[8] PhysiJS.js home page
http://chandlerprall.github.com/Physijs/

[9] JigLibJS2.js home page
http://www.brokstuk.com/jiglibjs2/

[10] Cannon.js home page
http://schteppe.github.com/cannon.js/

[11] Prall, Chandler 2012. JavaScript Physics Engine Comparison. *Build New Games*. 10 (Aug. 2012)
http://buildnewgames.com/physics-engines-comparison/