WebGLStudio – a Pipeline for WebGL Scene Creation

Anonymised

Figure 1. *Three screencaptures of WebGLStudio. Left-to-Right: Real-time editing on a highly realistic face asset; Post-processing effects (primarily Depth-of-Field) on a high-polygon model; Real-time editing of the parameters of a particle system.*

Abstract

We present WebGLStudio - a pipeline for the creation and editing of high-quality 3D scenes using WebGL. The pipeline features a 3D Scene-Graph rendering engine; an interactive scene editor allowing detailed setup and configuration of assets in real-time; a web-based asset manager; and a stand-alone rendering module ready to be embedded in target applications. We further present a series of implementation details necessary to overcome limitations of the web browser context to enable realistic rendering and performance. The principal contribution of this work to the graphics community is the methodology used to take advantage of several unique aspects of Javascript when used as a 3D programming language; and the demonstration of possibilities involved in real-time editing of the parameters of materials, shaders and post-processing effects, in order for the user/artist to create a 3D scene as desired.

CR Categories: I.3.7: [Three-Dimensional Graphics and Realism]; I.3.8: [Applications]

Keywords: WebGL, Web Integration, Real-Time Editing, HTML5, Pipelines

1. Introduction

Many digital production companies (whether for games, animated productions or digital film) now control and review their assets via web-based tools, due to the fast-moving ecosystem and rapid prototyping of web applications.

Professional applications such as Shotgun [2013] now provide browser-based tools which allow collaborative and remote working. Yet, despite using workstations with high-end graphics hardware, the inherent culture regarding browser-based and online 3D has meant that existing web applications do not make use of the full potential of the hardware in comparison to native (non-web) applications. This problem is exacerbated by a lack of a pipeline and tools for the creation, editing and export of 3D scenes for in-browser display. One reason for this lack of integration of Web-based 3D may be related to the challenges in attending to aspects of real-time rendering that go beyond the basic rendering of assets. Such challenges include:

- efficient local/remote management of large volume assets and data
- scene setup and editing using such assets
- correct export of render settings to ensure the scene can be viewed correctly in different environments.

Closed solutions such as Unity [2013] partially address these problems, but are also constrained by the requirement to use proprietary systems, and are principally designed for standalone applications (such as games) that are not intended to be integrated into wider systems. Thus, attention turns to more open, standard compliant methods for browser-based 3D. The first stage of such 3D technologies on the web is already complete. There are currently several libraries available, such as SceneJS and Three.js, to create 3D scenes on the web without having to struggle with the low-level layer of parsing, rendering and interacting with the scenes. Yet, these libraries are created for, and addressed to, people with a good knowledge of 3D graphics programming. They are intended to be the framework upon which developers can build their own 3D web applications, and thus are not so directly useful in the wider context of browser-based tools. Thus, there is a gap to be filled regarding to web/tool developers without 3D graphics knowledge who wish to integrate a 3D scene into their websites or web-based applications.

The solution to this second stage of 3D web technologies will come when the tools to construct and deploy 3D contents are mature enough to be accessible by all professional developers. One option may be to have some kind of standard markup language that allows the inclusion and editing of the 3D scene directly by the markup code. However, the majority of the variables of a 3D scene are too complex to tune without constant visual feedback, and some of them could easily 'break' the scene completely if they are tweaked without visual feedback. On the other hand, the users will likely visualize and interact on the web

browser, and a strategy of "what the creator/editor sees is what the user sees" enforced by a web-based editor, would solves a lot of issues that appear when turning non-web created scenes into web visualized scenes. A 3D scene editor, featuring a mature and powerful GUI, therefore becomes mandatory - as is exemplified in the wide-variety of scene editors designed for offline use, such as Autodesk Maya or 3D Studio Max. However, the problem with such editors is that they are designed to be used with their own proprietary offline rendering engines, so it can be difficult to match the scene successfully, even if a more standard format (such as Collada or FBX) is used. In this paper we present WebGLStudio, a pipeline designed to address all of these challenges, featuring principally a scene editor which allows real-time asset import and in-browser setting of parameters for materials and shaders.

2. WebGLStudio Overview

Our proposed solution to the issue of scene creation using browser-based 3D is a pipeline (and corresponding suite of tools) called WebGLStudio. The pipeline is based on our own 3D engine to edit a 3D Scene directly on the web, while the visual tool allows instant feedback of the final result. The tool has been created to make it straightforward to build, edit, and deploy a scene inside a working website with ease. The principal steps in the pipeline for the creation of a new 3D scene are:

- Import assets from several file formats (meshes, textures, materials)
- Visual arrangement of the entities in the scene (lights, meshes, cameras).
- Configure the visual appearance (materials, textures, meshes)
- Apply postprocessing effects
- Save the scene, export for external rendering

Once the scene is created all the information related to the structure is stored as a JSON file. To include it in a 3rd party website all that is required is HTML code that will load the JSON and render the scene on a HTML5 Canvas.

The entire pipeline is developed in a very *modular* fashion. One of the benefits of developing such a pipeline for the web is derived from the Javascript freedom to rewrite any element of code by importing another source file. As a result, the Pipeline is designed to be easily extendable using different modules (or plugins) that replace or extend parts of the pipeline. The editor itself (see below) only takes care of the interface and rendering, while the larger pipeline comes from the interaction between different modules. It is worth pointing out that this modularity would be more difficult to achieve in a language which is less dynamic than Javascript.

2.1 Core Engine

When developing a 3D graphics engine for the web, care must be taken to avoid efficiency problems related to the Javascript context (as mentioned above). We use glMatrix [2013] for all matrix operations because it has proved to provide good performance. For the lower layer (responsible for basic drawing and lighting) we use a heavily modified version of the library LightGL [2013] which wraps the common actions in WebGL with more user friendly classes (for shaders, textures and meshes).

Javascript is a very powerful dynamic language, which provides the developer with several interesting tools to overcome software design problems:

- The Prototype paradigm (the possibility to override existing methods and classes 'on the fly' i.e. in code executed at a point after the original)
- The facility to extend the structure with new components and function
- The ability to crawl an existing Object or Object hierarchy

While the upper layers of WebGLStudio (see other sections below) take advantage of all these properties, the core engine does so less in order to improve rendering performance. Every scene node is a component container (similar to the approach of modern game engines such as Unity). Every behaviour is wrapped around a component and can be attached to any node. Even cameras and lights are considered components. The components interact with the node, the scene and the rendering pipeline through events, so the pipeline does *not need* to have previous knowledge about their existence.

2.2 Render Pipeline

The render pipeline (which is wrapped into an independent module) reads the scene graph and renders a frame according to the information stored in the tree. We use a forward rendering solution instead of deferred solutions due to the lack of support in WebGL for the attachment of multiple frame buffer objects. The renderer supports any number of lights through multipass rendering (although naturally the performance can suffer when having too many lights). For shadowing the renderer uses PCF shadowmaps [Bunnell & Pellaci, 2004], and supports realtime cubemap reflections. For Normalmaps in tangent space the Renderer uses the derivative functions in the pixel shader, via the "OES_standard_derivatives" extension in WebGL, which is supported in Chrome versions 25.0 and above.

2.3 Editor

The most visually dominant aspect on the pipeline is the main Scene Editor which allows transformation and editing of imported assets. From the asset manager component of the pipeline (see Section 2.4 below), assets can be inserted into the scene, and position and scaled in real-time using tools and a free-roaming camera which will be familiar to any user of existing 3D software such as Autodesk Maya or 3D Studio Max (see Figure 2).

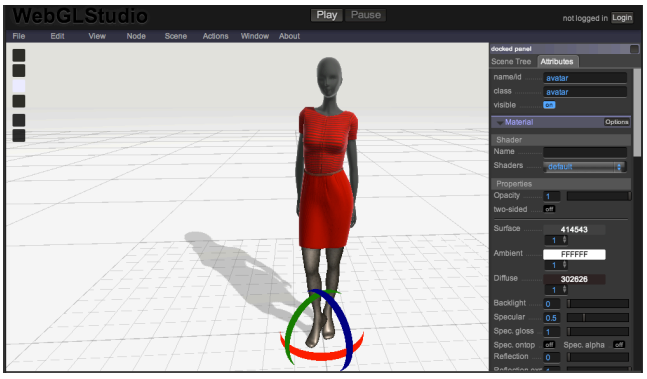

Figure 2. *WebGLStudio Editor, showing dynamic creation of editable attribute fields for a scene. The fields in the right column can be created dynamically by parsing the selected node in the scene tree.*

The editor instantiates the engine and dynamically creates editable fields and tools, based on the selected component in the scene tree. These fields allow direct and real-time changing of the parameters

of the 3D scene, thus enabling instant scene configuration without the need to edit code and reload the engine. The fields/tools are created by hooking events into the visual interface from the underlying engine. It uses a custom widgets library developed adhoc for this project, called LiteGUI. The concept behind LiteGUI is to use Javascript Prototypes to provide an underlying base structure for the GUI presentation of any scene component, while allowing a developer to fine tune the GUI with extra code if required. When the system creates the GUI for a component (e.g. the Particle System, see below) LiteGUI wil first check to see if the developer has coded a custom GUI layout/configuration for that component. If not, LiteGUI will fall back and create a 'default' GUI which will allow at least basic editing of the components properties.

2.4 Asset Management and Server Sync

When editing a scene the users need to have the freedom to import their own assets (textures, meshes, materials). Being a web application, there are two possible sources for assets, each of which presenting challenges for 3D scene creation:

1. *Local*: Web applications do not have native access to the hard drive for security reasons. Some HTML5 APIs such as localStorage or indexedDB address these issues but are designed to store small amounts of data (around 2.5 MB), while 3D editing may need around 100 MB for a full scene.
2. *Remote*: To ensure that other users can see the same scene as the one seen by its creator, all the content must be stored remotely on the web at some point. Furthermore, remote storage components allow further collaborative scene design.

The need to comprehensively address both use-cases has led us to develop a simple server file system library that can be accessed through a REST API. In this way, any client-side component in our platform can store data in the server. When storing the assets on the server, we use an indexed SQL database. This permits the system to track any changes, set privileges, store metadata, and enables users to add comments. We plan to extend our work in these points towards collaborative environments (see Section 4 below). The limitations of local storage are overcome by only saving scene structural information (see Section 2.9 below) which rarely approaches the size limit for localStorage.

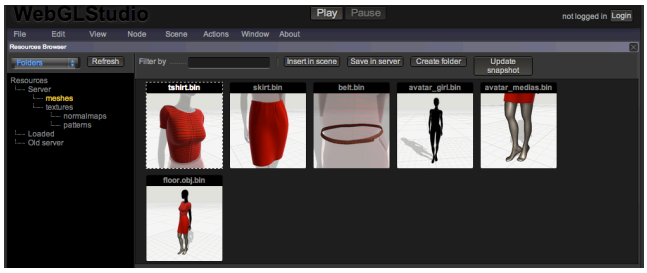

Figure 3. *The resource manager*

When storing the assets on the server, the data is optimized and recomputed in order to store additional data, such as missing streams, topological data, bounding boxes or octree information. This additional overhead when first uploading the asset frees the engine from having to compute the data at run-time, every time the asset is loaded into a scene. This is particularly advantageous when loading a scene with multiple assets, as the pre-computed information prevents the freezing of the browser process. To store the meshes in binary format we have created a Javascript library called BinaryPack that transforms Javascript objects containing Typed Arrays to a single ArrayBuffer that contains all the information, therefore storing the topology of the mesh in a GPU-friendly way. Thus, during the mesh loading process we do not need to parse any info; the data is already stored in typed-arrays so the upload to the VRAM is fast and non-blocking when using large meshes.

2.5 Post-processing effects

Post-processing effects are added to the output of the scene by using the classical approach of rendering to a texture and applying a shader when rendering to the screen. This approach has the same problem that occurs when working in regular OpenGL, which is the loss of Multisample Antialiasing (MSAA). To add antialiasing we use a supersampling approach, and then apply a sub-sampling shader when rendering to the screen. The current supported post-processing effects are color-correction, glow, brightness, contrast and depth of field. We are also working on using a modular graph (similar to hypergraph in Autodesk Maya or The Foundry's Nuke) to construct the final frame plugin, with different boxes representing image filters and effects.

2.6 Particle Engine

To test efficiency of Javascript updating and uploading a mesh to the VRAM we developed a particle system that permits configuration of several parameters in real time. Each particle system in the scene (containing hundreds of particles) is rendered using one single mesh, where every particle contained within it is an orientated quad. This mesh has to be updated and uploaded to the VRAM once every frame so ensure the particles face the camera. We were surprised to see very good performance (>60fps) even in systems with over 1000 particles. Even z-sorting the particles did not reduce the performance greatly (first we precompute the distance to the plane of the camera and use the Javascript *sort()* method). Some of the parameters that can be configured in the particle system are:

- Life, Size, Color
- Texture (we support animations though multiple frames texture)
- Alpha over time (using a curve)
- Size over time (using a curve)
- Physics parameters (gravity, speed, Friction)

2.7 Mouse Picking

In the creation of a scene editor, accurate and fast mouse-picking is naturally of high priority, in order to detect which object is selected when the user interacts with the 3D scene. Bounding box collisions were not considered suitable because of suboptimal performance when meshes are overlapping, or if the (free-moving) camera is navigating through them; and our goal was that the picking accuracy should be "pixel-perfect". The solution was to render the scene again in a separate buffer while assigning a flat color to every object. To speed up the process we use the OpenGL scissor test to limit the rendering to the pixels around the mouse. After that the pixels around the mouse are read using the gl.readPixels() function and the color is checked to see which object was behind the mouse pointer. This method has proved to be extremely effective because it only needs to be computed once per frame. It has very little noticeable delay, because most of the work is done in the GPU.

2.8 Real-time Mesh Painting

The Pipeline features a tool for realtime mesh painting that allows the user to paint into the texture, drawing directly on the mesh in the scene. To do this we need to solve two problems:

- Detecting the collision point of the ray that goes from the camera to the mouse.
- Paint the pixels of the texture that lay around the position where the ray collided the mesh.

To find the collision point we cannot use the same approach described for picking because we needed to know the exact collision point in world space. Instead, an Octree is created at the beginning of the painting process and is traversed as the mouse moves. Although, this is a more CPU intensive approach, the performance is sufficient for real-time use. Once the collision position is determined, our approach is to avoid simply translating it directly to texture coordinates and painting a quad in that position, because this approach would produce distortions in areas where the texture density changes dramatically. Our technique tries to locate which pixels of the texture lie within a certain distance of the collision point and colorizes them. To do this the mesh is rendered into the texture using the texture coordinates as vertex positions, and the original vertex coordinates as an extra stream. Thus, the mesh is effectively unwrapped over the color texture, and from the extra stream we have the pixel position in world coordinates. Finally, we can colorize according to the distance between the world coordinates of the pixel and the collision point. Because this solution is mostly computed on the GPU the results are instantaneous - with a mesh with more than 10,000 triangles the performance is good enough to paint on the mesh directly in real-time using a modest desktop system described below.

2.9 Scene saving and Export

The scenes can be converted to a JSON file containing relevant information about the scene nodes, such as material configuration and links to asset files. The serialization method of the scene sends a message to every node and component to retrieve the vital information and constructs a Javascript object that is converted to JSON using the JSON.*stringify()* method. This way we can use localStorage to save it or send it to our server to share it with other users. This JSON only contains data about the scene and materials, only references to meshes and textures are stored

3. Performance

The performance results presented in Table 1 are taken from a modest desktop system: Intel Quadcore 2.6 GHz, 3GB RAM, Nvidia Geforce 8600 with 512Mb VRAM. The browser used was Google Chrome v25.0. Our internal tests show rendering performance in Mozilla Firefox v19.0 to be 80% of that of Chrome.

Figure 4. *Scene 1 (see Table 1)*

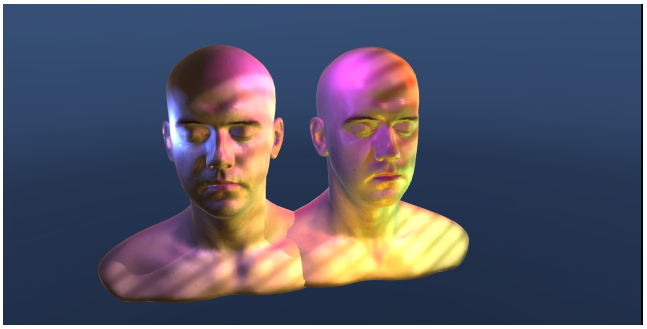

Figure 5. *Scene 2 (see Table 1)*

Table 1. *Basic performance, using texture resolution of 1024x1024, and screen resolution of 1400x700*

ID	Triangles	Details	FPS
1	30,000	1 light (1024px shadowmap)	60
2	40,000	Colourmap, Specularmap Normalmap; 8 lights (1 Shadowmap)	40

4. Conclusions and Future Work

In this paper we have demonstrated that WebGL is suitable for the creation of large, multifacet pipelines that enable rapid and real-time setup, creation and modification of 3D scenes. The principal contribution of this work is the methodology used to take advantage of several unique aspects of Javascript when used as a 3D programming language; and the demonstration of possibilities involved in real-time editing of the parameters of materials, shaders and post-processing effects, in order for the user/artist to create a 3D scene as desired. A secondary contribution of this paper is towards the democratization of 3D web technology, giving artists and non-technical users (for example, regular users of software such as Autodesk Maya or 3D Studio Max, who are familiar with concepts such as materials and shaders) the power to create high quality 3D scenes and distribute them via the web, without the requirement for proprietary plugins.

Our future work is focused on exploiting the inherent distributed nature of the web to allow collaborative scene development and editing. Current lines of work are to enable collaborative editing features more suited for the web (fork a scene, comment or tag information, make revisions...); enable realtime multiuser interaction for simultaneous editing of a scene; and create a modular editor for materials and behaviours. Our plan is to launch WebGLStudio as an open-source project, where the modular nature may allow it to be developed into a powerful tool for creation of web-based 3D scenes.

References

BUNNELL, M., AND PELLACI, F. 2004. Shadow Map Antialiasing. In *GPU Gems*, Chapter 11. Addison-Wesley Professional.

GLMATRIX. 2013. Retrieved Mar 11, 2013. http://glmatrix.net/

SHOTGUN.2013. Retrieved Mar 11, 2013. http://www.shotgunsoftware.com/

LIGHTGL. 2013. Retrieved Mar 11, 2013. https://github.com/evanw/lightgl.js

UNITY3D. 2013. Retrieved Mar 11, 2013 http://www.unity3d.com.