ReWeb3D - Enabling Desktop 3D Applications to Run in the Web

Tassilo Glander, Aitor Moreno*, Mauricio Aristizabal, John Congote, Jorge Posada, Alejandro Garcia-Alonso, Oscar Ruiz
Vicomtech-IK4, Spain - University of the Basque Country, Spain - CAD CAM CAE Laboratory U. EAFIT, Colombia

Abstract

Currently, 3D rendering is accessible within Web browsers through open standards such as WebGL, X3D, and X3DOM. At the same time, there is wealth of mature desktop software which comprises algorithms, data structures, user interfaces, databases, etc. It is a challenge to reuse such desktop software using the Web visualization resources. In response to this challenge, this article presents a novel framework, called *ReWeb3D*, which minimizes the redevelopment for migration of existing 3D applications to the Web. The redeployed application runs on a Web server. *ReWeb3D* captures low-level graphic calls including geometry, texture, and shader programs. The captured content is then served as a WebGL-enabled web page that conveys full interactivity to the client. By splitting the graphics pipeline between client and server, the workload can be balanced, and high-level implementation details and 3D content are hidden. The feasibility of *ReWeb3D* has been tested with applications which use OpenSceneGraph as rendering platform. The approach shows good results for applications with large data sets (e.g. geodata), but is less suited for applications intensive in animations (e.g. games).

CR Categories: C.2.4 [Computer-Communication Networks]: Distributed Systems—Client/Server; D.2.13 [Software Engineering]: Reusable Software—Domanin Engineering;

Keywords: WebGL, OpenGLES2, C++, JavaScript, Web, OpenSceneGraph, Software Migration, Mobile Devices, Client Display

Glossary

CSA	Client-Side Array
DCC	Digital Content Creation
GUI	Graphic User Interface
LLVM	Low Level Virtual Machine
LOC	Lines of Code
LOD	Level of Detail
OpenGL ES	OpenGL for Embedded Systems (mobile phones, video consoles, smartphones)
OSG	Open Scene Graph graphics Toolkit
TMS	Tile Map Service
WebGL	Web Graphics Library for 2D/3D interactive rendering.
XML	Extensible Markup Language for document encoding
WMS	Web Map Service

*e-mail:amoreno@vicomtech.org

1 Introduction

During over 50 years, extensive development of mature and robust software including graphic interaction has taken place, in domains such as medicine, physics, engineering, and entertainment. This software today runs mainly on desktop platforms, and only recently the challenge is being faced, of exploiting it in mobile devices such as smart phones, tablets, etc. (Figure 1). This evolution presents an urgent demand for multi-platform implementations. At the same time, Web technology and standards have an opportunity to face this challenge. Using standardized web technologies, unified applications can be developed which run on many platforms. With the recently added support of WebGL, web browsers natively support running 3D visualization applications [Marrin 2011].

However, for the wealth of existing desktop software with graphic interaction, migration to the Web represents several major obstacles: (1) Redevelopment entails significant mastering of languages and standards for web-related programming. (2) Millions of lines of code need to be ported. (3) Intellectual property must be protected if the source code transits the network. (4) Large amounts of transferred data reduce the performance of the 3D application. This article presents a prototype framework that provides a satisfactory answer to these issues for a significant application range.

The remainder of the article is organized as follows: Section 2 reviews the state of the art. Section 3 discusses the implemented methodology. Section 4 presents the implementation details. Section 5 describes the results obtained in the example executions and discusses the implementation performance. Section 6.3 draws the relevant conclusions and future challenges of our undertaking.

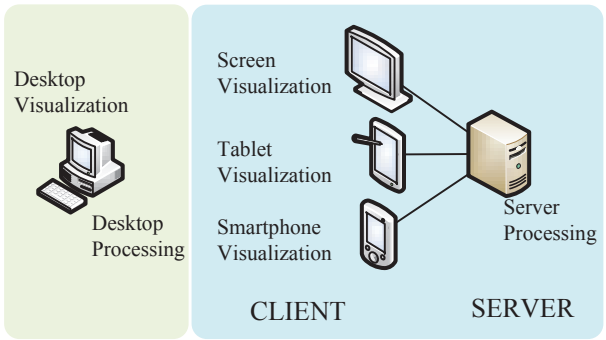

Figure 1: *ReWeb3D supports the migration of a 3D application from desktop to a WebGL-compliant client / server application.*

2 Related Work

A number of related methodologies exist, which address the general problem of migrating interactive 3D graphic desktop applications to Web browsers. These rendering methodologies are: (1) remote, (2) distributed, (3) pipeline, (4) plug-in based, and (5) WebGL. For the sake of clarity, in this article we will use the term *client* for the I/O system that directly interfaces with the user. The term *server* names the system that is remote to the system and usually carries the computational kernel of the application.

2.1 Remote Rendering

In *remote* or *image-based* rendering [Shi et al. 2009; Tizon et al. 2011], the rendering is completely calculated in the server. The generated images are continuously streamed through the network towards the client. The visualization occurs at the client, via a thin implementation. Mouse, keyboard or touch events are captured and sent back to the server in order to steer the application and subsequently to recalculate and refresh the scene. Remote rendering causes a high network load. This network load is, however, relatively constant because the continuous data flow only depends on the canvas resolution. Remote rendering requires high speed connections to avoid network saturation even if optimized and/or compressed transmission methods are applied.

2.2 Distributed Rendering Pipeline

A Distributed Rendering Pipeline runs 3D applications shared between client and server by using transferred-drawing calls [Humphreys et al. 2001; Magallón et al. 2001; Neal et al. 2011]. A classical example of distributed rendering is Xgl on Linux or Unix platforms. In general, the 3D application is run at the server, while low-level (graphics API) drawing calls are dispatched and executed at the client by using its local resources. Interaction events are captured at the client and sent to the server for remote processing and adaptation of the drawing commands (scene refreshing). It must be noticed that distributed rendering requires a fast connection and tolerates lower delays, as it must be synchronous.

2.3 Plug-in Based Methods

A plug-in is a binary application (e.g. 3D visualization) installed at, and subordinated to, the web browser. A plug-in generally has full access to the resources of the client, hence increasing the flexibility of the implementation. The plug-in may contain (1) a significant computational kernel (the migrated application) which therefore remains foreign to the browser itself and (2) a visualization part. Web 3D interactive visualization plug-ins are located in, and use, the client hardware. Commercial examples are Unity3D or Adobe´s Stage3D. Major disadvantages of the Web visualization plug-ins are: (a) lack of standardization across browsers, (b) need of manual installation and maintenance efforts, (c) network security risks.

2.4 WebGL based approaches

WebGL is a low-level 3D API for web browsers. It was specified as an open standard by the Khronos Group in 2011 [Marrin 2011]. WebGL has been adopted by most web browsers. Its version 1.0 is a JavaScript binding of the OpenGL ES 2.0 API (e.g. a subset of Desktop OpenGL) depending on vertex and fragment shaders and natively running on the GPU hardware.

Despite WebGL being a relatively recent standard, there are several browser-based 3D middleware frameworks supporting WebGL as rendering back-end. Examples are SpiderGL [Di Benedetto et al. 2010], X3DOM [Behr et al. 2011; Behr et al. 2010] and XML3D [Sons et al. 2010]. X3DOM addresses the specification and manipulation of the 3D scene using X3D and a document-object model (DOM) schema for the access. X3DOM is used widely, for example, as part of a web-based DCC pipeline [Ulbrich and Lehmann 2012]. XML3D aims at a minimal addition to standard web technologies to enable 3D content.

Migration of already established desktop 3D applications into JavaScript is a highly demanding task due to the differences between programming languages and to the amount of code to be ported. As a response, the Emscripten compiler has been implemented, which is able to compile C/C++ code into Javascript, using LLVM bytecode as an intermediate step [Zakai 2011].

These first attempts show promising results with ported 3D engines. A significant standing problem is that each library used in a 3D application has to be migrated, downloaded and run by the client.

3 Methodology

This section discusses the technical milestones of the ReWeb3D framework. WebGL-capable browsers (Figure 2) are novel rendering platforms which enable standard-based, plug-in-free 3D Web applications. This capability is relevant in the context of reusing large portions of existing desktop 3D applications. As a case study Figure 2 shows an example (OSG Earth) running on OpenGLES2, which provides an API for multiple rendering platforms. Using ReWeb3D, desktop 3D applications can additionally use WebGL-compliant web browsers as a target platform.

We will discuss the methodology of a novel framework to migrate desktop 3D applications to web pages, based on a client-server distributed computing scheme (Figure 1). Opposed to running both, the application and its GUI in the desktop, we implement a framework in which the application is run (and its data stored) in the server, while its visualization is run in (possibly) several clients.

The framework is implemented as follows:

1. High-level structures and objects of a visualization scenario (parametric curves, surfaces, polyhedral shells, text) are converted into low-level graphical primitives (e.g. triangles, lines, and points) by the visualization application.
2. Calls to the low-level graphic application programming interface (API) are then redirected to a web application server.
3. At the server, a web page is generated, containing the corresponding browser code.
4. The Web page is sent to the client.
5. At the client hardware, the web page is loaded and executed, performing the rendering procedure.
6. Interaction events for the camera manipulation of the scene are tracked and handled at the client, to avoid network transit.
7. Dynamic scenes are computed at the server.
8. Animations or scene regenerations are sent through asynchronous JavaScript and XML (*AJAX*) updates.

This methodology has several advantages: (1) The strengths of low and moderate dynamic graphic application are preserved. (2) Only minimal additional development is needed to connect with the web application server. (3) Only the low-level graphics API calls are redirected, serialized, and sent to the client. (4) No high-level source code is given to the client, therefore protecting the intellectual property of the application. (5) The network bandwidth is used efficiently, as only necessary updates are requested through the network. (6) Distributed computation of the scenes balances the computational work load between server (e.g. storage, traversal of the scene graph, culling [Akenine-Moller and Haines 2002]) vs. client (e.g. event handling and low-level rendering on the GPU). The downturns are that (a) highly dynamic scenes are not preserved and (b) specific interaction techniques would still need to be manually ported to JavaScript

Figure 3 shows the general architecture of our approach. We replace the GL driver by a GL proxy that captures calls to the graphics card (GPU) and embeds them in a WebGL canvas of a web page. The

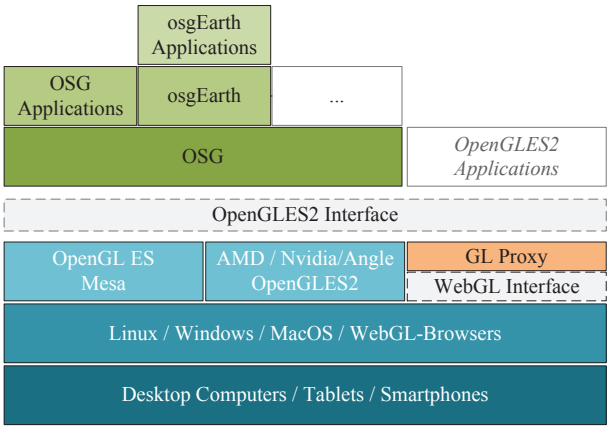

Figure 2: *OpenGLES2 Interface supporting the migration of desktop applications. This example addresses an* `OpenSceneGraph Earth` *application.*

client loading the page executes the WebGL calls (possibly with hardware-accelerated graphic functions). The desktop 3D application is embedded in a server environment. The rendering is not executed on the *server*. Instead, its visualization calls are converted to a 3D web page and its low-level rendering commands are executed in the *client*'s browser. The application itself (in particular sensitive code) remains largely on the server. If a web page serving as the entry point of the application is requested by a browser, the web page is created by our framework based on these major considerations:

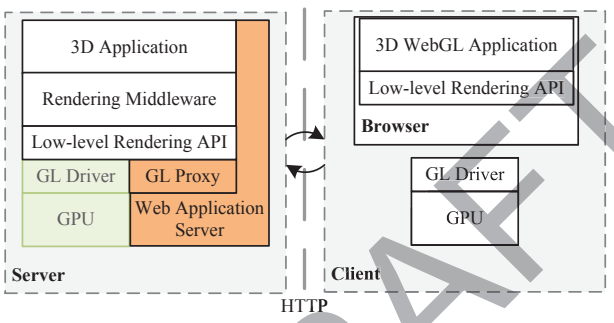

Figure 3: *General architecture. Visualization activity embedded in a WebGL canvas, executed locally at the client.*

1. We provide a thin software layer called *GL proxy* which uses the low-level rendering interface (Graphics Library, GL) to capture GL commands and to generate the equivalent browser commands.
2. Client and server are asynchronously connected.
3. The client renders the scene independently and only gets occasional scene updates from the server.
4. The GL commands based on their individual semantics are classified into commands used for (Figure 4): (1) initializing (`Init`), (2) continuously (re) painting (`Paint`), and (3) occasionally updating (`Update`) the 3D application.

Figure 4 shows that in classical desktop 3D applications, calls to the low-level rendering API are made by the application at the server (which is also the client). In contrast, our approach intercepts these low-level visualization calls, generates appropriate JavaScript code and sends it to the client for execution. Hence, repeated execution of renders is moved from server to client. Only at regular time events, the client gets back to the server to update the current scene.

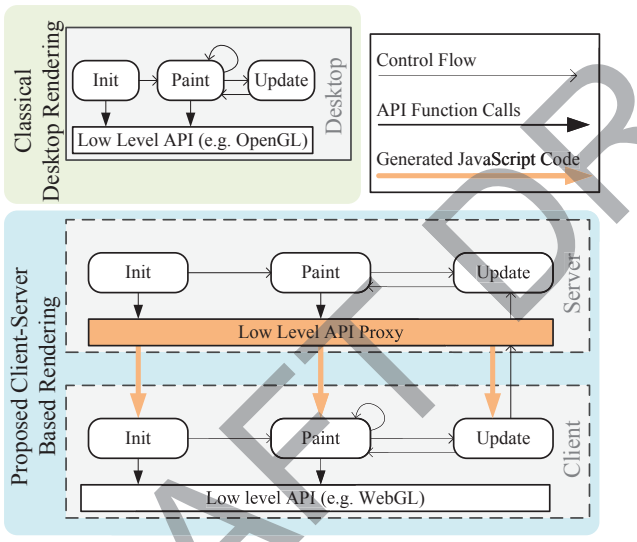

Figure 4: *Comparison of visualization in desktop applications against server - Web client visualization.*

3.1 Distributing the Rendering Pipeline

When the kernel of the application is geographically removed from the geometry and rasterization [Akenine-Moller and Haines 2002], the commands transit the network. For repeated generation of rendered images (render loops), the GUI commands must be sent repeatedly over the network. This transit would negatively impact the render performance.

In contrast, our framework reduces network utilization and dependency between client and server by taking a snapshot of one iteration of the rendering loop on the server. This snapshot is then sent to the client, which locally performs the actual rendering loop by repeatedly dispatching the low-level commands. This scheme distributes the application part of the rendering pipeline between client and server.

To accomplish this process, we intercept the graphics pipeline at the interface to the low-level rendering API, which is normally invisible to the application. The only change to the application is the way in which the rendering is triggered. Conceptually, instead of nesting the rendering in a loop, each low level GL command executed during rendering has to be called only once to be able to capture it. Embedded in the server, the GL commands are then generated as Web page source code. The client receives this source code inside a minimal 3D application, which only executes the low-level commands in the render loop and handles events either locally or by sending them back to the server.

3.2 Organizing GL Calls into Stages

Efficient rendering implementations depend on organization of GL functions into functional blocks, which we call *stages*. Typically, there is an `initialization` stage, which is called first to setup geometry, attributes, or textures. The `paint` stage is repeatedly called to actually draw the scene, thus enabling continuous animation. This organization is also supported by many window managers or GUI toolkits, which provide hooks to set callbacks to these

functions. We adopt this organization for the minimal 3D web application which runs on the client. To do so, we manage an internal state in the GL proxy by specifying the current functional block as one of `Init, Paint, Update`. Before calling the respective function on the server, the state must be set. Subsequent low-level API calls are captured and stored in the respective function block for the client application. If the desktop 3D application has the same organization into functional blocks, this capturing can be implemented in a straightforward way by setting the state of GL proxy just before proceeding with the usual function block.

However, some desktop 3D applications and middleware framework provide only one central aggregated render function or `frame()`, instead of separated functions. In such cases, the application manages the rendering resources by initializing them on the first reference, and by using them on subsequent references. For a unified client implementation, we aim to enable the same organization into stages as before. In our approach, we filter GL functions depending on their semantics.

For filtering we classify all GL functions as belonging to one or more of 3 stages:

- The `Init` and `Update` stages contain GL functions that are used to setup the relatively static parts of the rendering application. Examples are the creation and fill of geometry buffers, texture data, and shader programs.
- The `Paint` stage contains GL functions that are to be executed in the rendering loop. Examples are the drawing functions, binding the current buffers, textures, shaders, and setting of shader parameters.

Having the classification of all functions from the GL interface, we can filter the functions according to the internal status of our GL proxy. As shown in Figure 5, we call the central rendering function of the middleware (e.g. `frame()`), several times with the GL proxy's state set to different stages (e.g. `init`, `paint`). After the server is set-up, when query from a browser appears, the `init` stage is set before calling `frame()` to catch all GL calls relevant for preparing the rendering. Then, we set the state to `paint` and call again `frame()` to catch the GL calls to do the actual rendering. Then, the generated WebGL page is sent to the client for execution on its WebGL-compliant browser. The browser in turn initializes the rendering and enters the render loop repeatedly, executing the `paint` calls.

If a change of the scene occurs, the browser queries the server for an update. Such changes can be triggered by larger movements of the virtual camera or by interactions with the user interface. The current status of the browser is sent to the server (e.g. the virtual camera position or pressed buttons). This status is entered as a sequence of events to the attention of the server application. To generate the necessary update, the server sets the proxy GL stage to `Update` and reruns `frame()`. The created JavaScript calls are sent to the browser and executed once, adapting the render loop to reflect the changes.

Summarizing, the server updates the scene at relatively low frecuencies. Each update at the server generates the necessary WebGL code. The actual render iterations creating images at interactive rates takes place solely on the client. This approach decouples server and client in the rendering pipeline.

4 Implementation

To test our concept, we have implemented the framework, using `OpenSceneGraph` as a 3D middleware and Emweb `WebToolKit` as the web application server. `OpenSceneGraph`

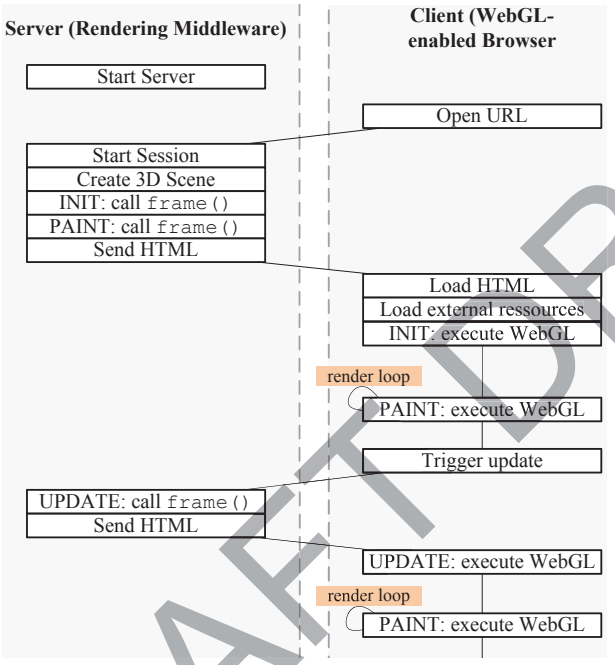

Figure 5: *General sequence of actions: the server runs the 3D application to capture GL calls corresponding to the `Init` and `Paint` stage, and sends the HTML page to the browser. The browser executes the render calls, enabling local interaction. If larger updates are necessary, the changed GL calls are generated within the `Update` stage on the server and sent to the client.*

is a widely used open source rendering engine of considerable functionality and active developer constituency, which supports OpenGLES 2.0. `WebToolKit` is a framework to build web applications with C++, allowing creation and service of complex web pages, which has basic support for WebGL.

4.1 Application Programming Interface - API

The API chosen is a subset of desktop GL, corresponding to the standard `OpenGLES 2.0` (GLES2), developed for embedded 3D applications [Aaftab Munshi 2010]. The reasons for this choice are: (1) WebGL is based on GLES2 and thus very close in terms of syntax and semantics. (2) GLES2 implements a subset of desktop OpenGL, enabling high portability of applications for desktop. (3) GLES2 has dropped the fixed function pipeline to keep the interface and hardware implementations small, in comparison to OpenGL 2.

The choice implies that applications need to provide custom shader programs, manage a matrix stack if needed, and cannot rely on immediate mode rendering. These consequences are congruent with the evolution of OpenGL versions beyond 3.1, where the fixed function pipeline is restricted. GLES2 does not include more advanced features available for the desktop, including 64 Bit floating point precision, geometry and tessellation shaders, and in general, has reduced resources.

To summarize, GLSE2 is a small but capable subset to GL functions, which enable multi-platform development of applications for (1) desktop, (2) embedded systems, and (3) Web applications, minimizing the need for adaptations and learning of new interfaces.

4.2 GL Proxy Embedding

`OpenSceneGraph` provides `frame()`, a central method to render a single frame of the scene with the current view settings. Within this method, the scene graph is traversed to perform view frustum culling, state sorting, conversion of geometry into low level primitives, etc. The optimized scene description is sent to the graphics driver to perform the rendering. As part of the rendering, `frame()` will be called repeatedly to produce interactive 3D graphics. The iteration is controlled by a timer, to have a desired refresh rate or with maximum speed, to saturate the refresh rate.

To implement our concept, we need to intervene the existing pipeline at two places:

- `OpenSceneGraph` needs specific code to interact with the GUI toolkit (e.g. receive events, trigger the rendering of a scene frame, provide a canvas window). We replace this toolkit specific code, since we need to control calls to the central rendering function.
- We replace the library that implements GLES2 by our GL proxy library. Hence, we receive all API calls to the graphics driver and apply custom implementations for them.

The first alteration is necessary, as we do not need continuous rendering on the server-side. Instead, we manually trigger `frame()` explicitly. When generating the Javascript code for the respective `Init`, `Paint`, and `Update` stages, we first set the state of the GL proxy to the respective state before calling `frame()` once. In this manner, we can decide, based on the current state of the GL proxy and the semantics of the GL function, whether to generate Javascript code for a specific function or to omit it.

The second alteration is the replacement of the GL driver by the GL proxy library, which receives the GL calls. In the GL proxy, the majority of the GL calls just need to be relayed directly by writing the appropriate Javascript string into the output script. Examples of these calls are `glClearColor()`, `glBlendFunction()`, and `glLineWidth()`. In case of GL objects that are created for later reference (buffers, textures, shaders and programs) the GL proxy library makes the client to store these in Javascript variables and maintains a mapping between client-side variables and server-side objects (GL object identifiers) (Figure 6, left).

Regarding GL calls that query the state of the client context, we assume:

1. Constants such as the maximum number of vertex attributes, texture units, or available extensions can be queried on initialization and are reported to the server.
2. Efficient middleware implementations such as `OpenSceneGraph` maintain the GL state internally by tracking changes (e.g. last texture, color, and buffers) to avoid redundant calls to GL. Therefore, we do not implement the respective querie functions in the GL proxy. Exceptions are some shader related queries, as discussed later.

For a number of GL calls, the implementation is not straightforward. We discuss them next.

1. Geometry data including vertices, colors, normals, texture coordinates, and indices is provided in the form of buffers which are allocated and maintained by the graphics hardware. We serialize and aggregate the data in binary blobs to be sent to the client.
2. To support CSAs, which are specified in OpenGLES2 but not in WebGL, temporary buffers are created, filled, used, and discarded within the `Paint` function. Pointers to CSAs are also maintained, along with their *enabled* status (Figure 6, right). The temporary buffer option is not optimal since the data needs to be transferred to the graphics card for each frame. After appraisal of the pros and cons, we still choose to support CSAs and to issue a performance warning when used.
3. For setting up shader programs and relevant uniform and attribute data, middleware implementations rely on queried values. An example is posed by *uniform variables*: Only uniform variables reported as *active* need to be bound. To return active uniform variables and attributes without actually compiling the shader on the client (which requires a round-trip to the client), we parse the shader source for present variables during the `Init` phase and return the results, when queried.
4. For textures, we create a temporary raster image in memory, provide a URL, and call the WebGL function with a URL link to that image.

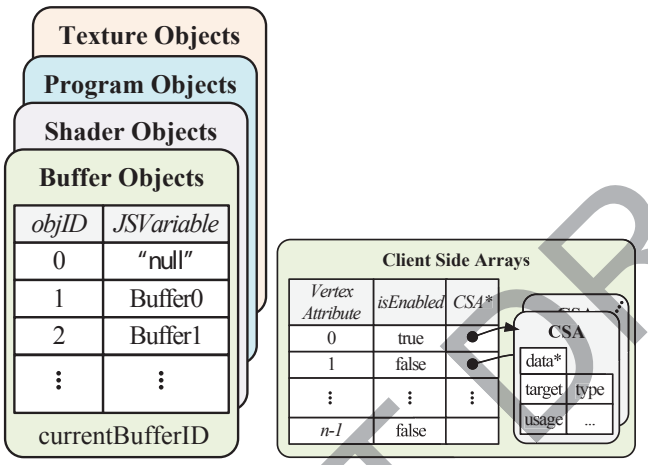

Figure 6: *Left: The GL proxy maintains the mapping between client-side and server-side GL objects. Right: In a table, the GL proxy stores for all n possible attributes of a vertex, whether a client-side array (CSA) has been set. Disabling an attribute array still keeps the CSA for potential later re-enabling.*

4.3 Camera Control

In addition to static aspects of a 3D scene, the users must be able to explore and navigate the scene. To support decoupling client-server and enable interaction without continuous roundtrips to the server, we provide custom interaction to the client. In it, the mapping from input events to virtual camera changes is confined to the client. A Javascript Matrix variable is administered, which represents the view transformation dictated by the input events. To apply the extra transformation, we need to inject an additional matrix uniform variable to each vertex shader code to multiply it to the output vertex. We assume that the rendering middleware applies transformations by using a combined model- view transformation matrix $T_{all} = V_{server} * M_{n-1}...M_1 * M_0$. The combined model-view transformation will undo the server-side transform and then apply the client-side transform $v_{camera} = V_{client} * V_{server}^{-1} * T_{all} * v$. This approach ensures the correct transfer to the browser of potentially nested transformations typical of `OpenSceneGraph`, while keeping smooth interaction. It must be noticed that camera manipulation still has to be reimplemented on the client-side. By default, we provide the client with basic trackball manipulation.

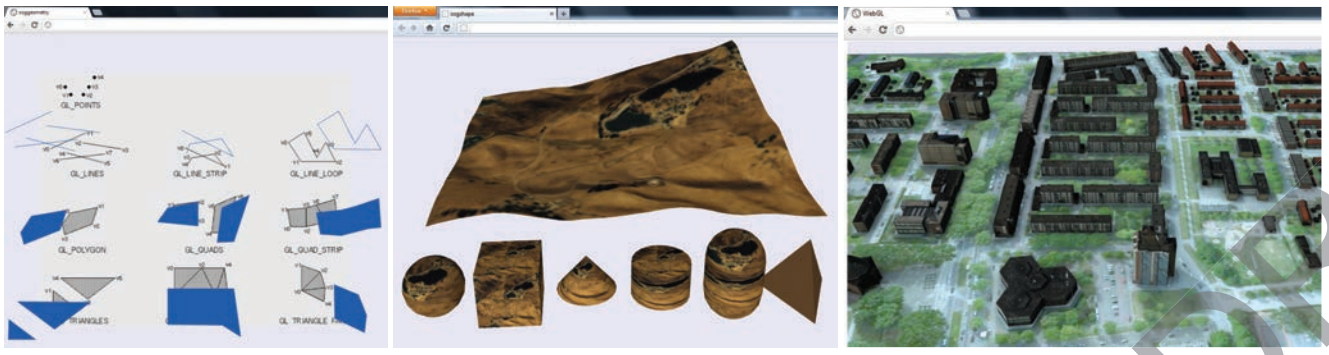

Figure 7: *Basic examples ported from* `OpenSceneGraph`*:* `osgGeometry`*,* `osgShape`*, and* `osgViewer` *loading a CityGML urban model. Colors and textures have been adapted to enhance legibility.*

5 Results

5.1 Basic Examples

We demonstrate our implementation with basic `OpenSceneGraph` example applications, such as `osgGeometry`, `osgShape`, and `osgViewer` (Figure 7). They illustrate the rendering of different primitive types, textures and automatic tessellation of analytical shapes.

The dataset used is a portion of the Rotterdam virtual model (publicly available at `http://www.rotterdam.nl/links_rotterdam_3d`) in the `CityGML` standard format (Figure 7, right). Notice that geospatial models typically contain protected data that cannot be distributed freely, are stored in specific data formats, and are massive.

To run the examples, we have to load our default shaders to the scene graph, since the examples depend on a fixed function pipeline. `OpenSceneGraph` already provides vertex attributes for geometry as well as the transformation matrices using standardized names which we can use in the shader. Camera interaction was kept confined to the client.

5.2 Rendering of 3D Geovirtual Environments

As geovisualization examples, we chose rendering of 3D geovirtual environments based on `osgEarth` and `VirtualPlanetBuilder`, which are terrain rendering toolkits for C++ desktop applications. Our example displays a virtual globe with different textures applied (`OpenStreetMap`, aerial photos) as well as a textured local terrain model (Figure 8).

The examples include massive raster and geometry data that has been preprocessed to multiple level of detail (LOD). Using a configuration protocol, data sources with different spatial reference systems can be specified for the current scene. This diversity allows the coexistence of different terrain and texture layers, features, and labels. Data sources may be files or web services (e.g. WMS, TMS). At rendering time, the engine selects the geometry and surface data for the current view point, applies optimization, (e.g. culling, state sorting), and sends the low-level commands to the rendering subsystem.

The library `osgEarth` is one for which a rewrite would be non-trivial. In contrast, in ReWeb3D the system runs on a server and writes to a WebGL-enabled web page to be executed in the client browser, render the data (e.g. globe). Only data in the appropriate LOD is transferred and rendered on the client. To enable streaming of new scene data, the thin client sends, for example, its current view parameters and requests a new frame in configurable intervals (e.g. 500 msec). The ReWeb3D wrapper adapts the server camera and re-evaluates the scene graph in the `Update()` and `Render()` stages, sending the updated scene data and the rendering commands to the client.

5.3 Applications Running on Multiple Platforms

Providing a WebGL client, our framework offers access to the visualization application for various platforms. Specifically, we target desktop applications, compatible browsers (Figure 7) and in general, any device capable of WebGL. We have tested the performance of the applications, modified as discussed, on mobile devices. We have chosen different operating systems, displays and resolutions to test the urban model (Figure 9).

In the tested ReWeb3D prototype the loading time and the achieved frame rate are far from being practical. However, the central result is that the identical, candid, examples are used in the tests and not versions specifically crafted for mobile devices. Regarding the current slow response, we argue that (1) the ReWeb3D framework is able to configure the middleware to deliver lower LOD scenes when running on mobile devices, and (2) increasingly powerful devices make practical the prototypes that are slow in the current hardware.

5.4 Performance

The performance of the current implementation was tested in Google Chrome 25 and Mozilla Firefox 18 browsers running in a desktop PC. Both browsers natively support WebGL. By the time of writing this paper, Microsoft Internet Explorer and Opera do not support WebGL natively. On Windows, Chrome and Firefox browsers use Google `Angle` library to translate WebGL/GLES2 to Microsoft `DirectX9` for better driver support. Since Firefox allows to use native OpenGL calls, we tested performance both for native and for `Angle`-based implementation.

The tests for this article were conducted using an Intel Quad Core Q9400 processor, 4 GB of RAM and a GeForce GTX 285, Windows 7 64 Bit (Service Pack 1) with the latest stable graphics drivers. Both client and server are in a local network, but exposed to full Internet traffic. We measured (1) the amount of data downloaded, (2) loading time, (3) memory consumption, and (4) effective frame rate at the client for the examples above.

For all tested browsers, the resulting frame rates are high (above 50 FPS), and in the basic examples the downloaded data size is below 200 KiB, taking less then 500 msec. to start the rendering. However, runtime memory consumption differs strongly across the

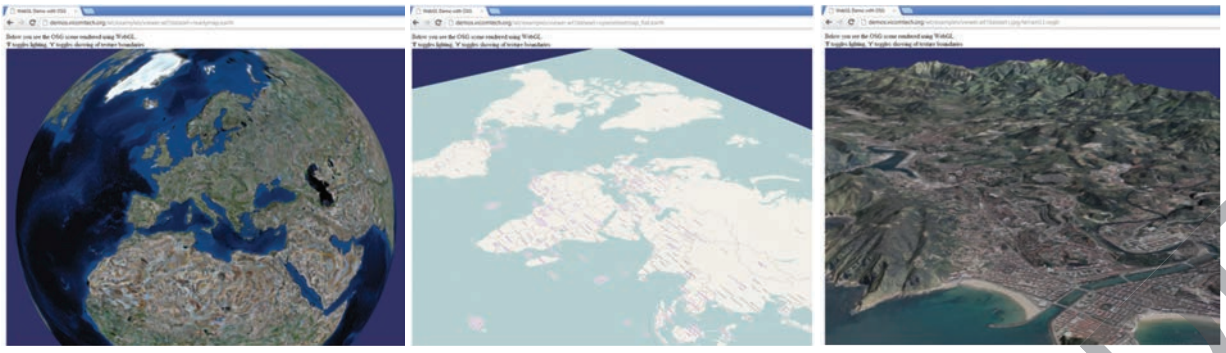

Figure 8: *Left: Global spatial models with spherical projection and color texture. Center: flat projection and OpenStreetMap texture. Right: Multiscale terrain model with color texture.*

browser. This variation obeys to the differing status of the support for WebGL in each one.

The urban model (thematic, geometric, appearance model) had approximately 30 MB. The processed model sent to the client contained a texture atlas with reduced resolution and batched geometry of 8 K vertices, demonstrating the practical use of a middleware. The downloaded data is only about 6.7 MBs, resulting in a remarkable loading time of about 7 sec. across all platforms.

In large scenes, such as 3D geovirtual environments, small lags may occur as large objects are coming in and out of view and have to be transferred to the client. `OpenSceneGraph`'s mechanism to control the update rate needs to be adapted to reflect the smaller bandwidth.

For server-side animation, we tested a simple rendering of a rotating triangle, where the changed matrices are sent repeatedly to the client. While the frame rate of this simple rendering is still very high, the amount of data sent regularly is about 270 B, downloaded in about 5 ms., each. As a 4×4 32-bit floating point matrix has only 64 B, this performance indicates a considerable overhead for the `AJAX` calls and points out a future work domain.

5.5 Porting Use Case: A Fire Spread Simulator

We selected the Fire Spread Simulator, an `OpenSceneGraph`-based application, and ported it to our framework. The goals are the exercising of the porting process, the identification of possible pitfalls, and the estimation of the working hours to be invested. The Fire Spread Simulator is part of a Ph.D. research and comprises around 8.000 lines of code.

The software simulates fire spreading in forest and urban areas using algorithms based on a cellular automaton [Moreno et al. 2011]. The algorithms are designed to be used at interactive rates. The user can interact with the fire by playing scenarios (e.g. hinder, start, limit). The simulator consists of Simulation, Visualization, and User Interface components. The goal of our porting is to run the simulation and to visualize components on the server while showing the interactive rendering on the client.

5.5.1 Application Structure

The static topographic data includes a digital elevation model (DEM) and classification of the ground in disjoint zones. Models for buildings are specified in XML files, storing material, height, and number of floors. Internally, a grid of cells represents the scenario for the fire spread model. Each cell records the local status and local fire intensity (among other data). The algorithm iteratively updates a corresponding colored image, where each pixel represents the status of a cell.

Figure 9: *A city model test running on 3 mobile devices: (i) Apple's iPad 1. (ii) Samsung Galaxy Tab and (iii) Sony Ericsson Xperia neo V mobile phone.*

In the visualization component, the scene graph comprises the terrain model with the aerial texture and the fire model texture blended on top. In addition, procedural geometry for building and tree models is added to the scene graph.

5.5.2 Porting Considerations

1. The main function of the application is changed to an initialization method which is called when a client connects the Web server. This method launches a WebGL server and prepares the basic static data.

2. The command line arguments of the original application are provided to the WebGL via a configuration file or URL encoded arguments. In this case, a XML file was prepared and loaded.

3. Due to WebGL restrictions, the textures must be uncompressed. In this case study the textures were the aerial terrain view and fire ones.

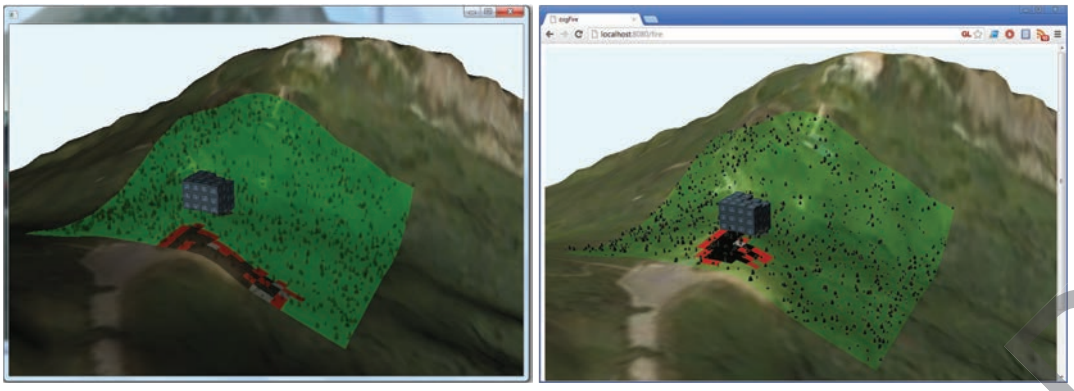

Figure 10: *The Fire Spread Simulator running as a desktop OpenGL application (left) and as a migrated client / server WebGL application (right). Minor color differences are due to differences between fixed OpenGL and our own WebGL shaders.*

4. The simulation at the server is decoupled from the render loop. The simulation runs in continuous manner on the server, but only runs the render method when triggered by the client.

5. Textures in the web application are implemented with customized shaders. They compute the texture coordinates and correctly place it on the terrain texture. The overlay color is blended with the color from the aerial view of terrain texture and written as fragment output.

6. While the fire simulation is running and continuously updating the overlay image, the `OpenSceneGraph` framework makes sure the texture is updated accordingly. In consequence, also the updated images are transferred and applied at the client.

7. `OpenSceneGraph` resizes our overlay image to Power-of-two (POT) sizes. This resizing produces minor resampling artifacts.

8. The tree models contained in the scene are rendered using a billboard technique. Their individual vertical rotation is adapted (per frame) to face the user camera, and the `OpenSceneGraph` computes the aggregated model-view matrix on the CPU. This process requires a large number of Uniform matrix variables to be updated. As a consequence, the network traffic is very high and the correct orientation is only obtained after an update from the server.

5.5.3 Case Study Results

Figure 10 shows a comparison of the desktop simulator and the achieved result through the ReWeb3D framework. The implementation of the fire spread algorithm and the source data used for calculations remain in the server. The result of the porting process is considered a very promising one. In less that 15 working hours we managed to port large parts of the functionality of an `OpenSceneGraph`-based desktop simulation comprising about 8.000 LOC to the Web. A traditional porting of the application would require resources enlarged by several orders of magnitude.

Some of the issues reported required changes to the application, but do not change its functionality. On the other hand, forcing the splitting between the kernel of the application and its graphical output would benefit not only web-based but also desktop-based applications. Decoupling client-server responsibilities gives, for example, the opportunity transfer CPU operations to more efficient GPU implementations. However, separating the implementation also adds complexity when supporting dynamic scene evolution. Considerations associated with the use of fixed-functionality and many updates are: (i) `OpenSceneGraph` classes currently relying on fixed functionality (e.g., texture coordinate generation, texture blending, lighting), are expected to be replaced by shader-based classes in the future. (ii) Rendering mechanisms depending on CPU side computations (e.g. billboard orientation and client-side arrays) have to be customized upon migration in order to avoid bandwidth problems.

6 Discussion and Conclusions

6.1 Technical Discussion

We have presented a successful prototype of the proposed methodology to migrate `OpenSceneGraph`-based desktop applications with high performance and graphical quality on browsers and mobile devices. The example takes advantage of the middleware functionality in: (1) use of importers for images and geometry, (2) tessellation of analytical shapes, (3) preparation of low-level primitives, (4) definition of the scene graph, and (5) status sorting for efficient rendering calls.

With our methodology, legacy applications can be rapidly migrated if they do not rely on the fixed function pipeline. If they do, default shaders may provide similar functionality. In middleware such as `OpenSceneGraph`, setting up parameters is done for transformation matrices, but not for all fixed-function attributes. Another drawback of choosing OpenGLES2 as API is its limitation to widely supported functionality, which excludes some more recent features of graphics hardware.

Uncoupling server and client obviously is a non-trivial task. If scene changes imply small data sets, the transit in the network is kept down. For example, if basic camera control may be provided only on the client, the server could be ignored for many refreshing purposes. On the other hand, scene changes requiring massive data computing in the server (e.g. complex camera navigation techniques, particle systems, physical simulations) imply considerable, and expensive, server-client communication.

6.2 Relevance for System Integration

A formal design of experiments to evaluate efficiency in code migration (as result of our approach) would be impossible. Instead, we discuss the expected issues and problems when undergoing the migration:

(1) Existing desktop applications are a repository of know-how, money and time of very large proportions. Their migration to

the Web cannot be ignored, but at the same time, it is extremely expensive in training, time, money, etc. As examples, the libraries used in the Results section have the following sizes: (i) `OpenSceneGraph`: 300K LOC and (ii) `osgEarth`: 200K LOC.

(2) After migration to the Web, applications are to be maintained and improved. Decoupling server and client parts is positive for both, desktop and web applications, because by sharing code it lowers the probability of errors in the maintenance and development.

(3) In our approach, low-level rendering calls, shader programs, and primitives are exposed in the network, while the high-level functionality remains on the server. Original source code as well as original model assets are not transferred over the network, therefore protecting intellectual property.

(4) High performance computing (CUDA, OpenCL, etc.) is not yet supported in mobile devices. With ReWeb3D, high level computations are performed on the server while low level rendering commands are executed on the client.

6.3 Conclusions

Our approach complements other fundamental approaches to web based 3D visualization, e.g., WebGL standalone applications and image based rendering.

Applications with a focus on preprocessing, relatively static data, and implementation of rendering effects done on the GPU can be migrated with low effort. Upon execution, analysis and processing can be conducted on the server. Only the visual results have to be transferred.

In contrast, applications relying heavily on fixed functionality and OpenGL 1.2 / OpenGL 2.0 immediate mode rendering or highly dynamic data are difficult to migrate. Support for these APIs by the framework would require major implementation effort because they either require additional client-side implementation, or repeated communication between client and server (e.g 3D games).

6.4 Future Work

Future work includes: (1) investigating the use of binary exchange formats and compression for geometry data, (2) increasing debugging capabilities, (3) balancing workload between client and server, (4) planting small pieces of code in the client at runtime, for added, locally executed functionality. (5) streaming data from external sources, map tiles, video, directly to the client without the need to pass through the server. (6) enabling client-side, dynamic animations, with the client receiving a high-level scene representation and considerable additional implementation, based on existing code on the server-side.

Acknowledgements

This work was partially supported by the Basque Government (ETORTEK research project ITSASEUS II), the COST Action TU0801 "Semantic Enrichment of 3D City Models for Sustainable Urban Development", and involved additional collaboration from invited researchers from the CAD/CAM/CAE Laboratory at Universidad EAFIT, whose internship at Vicomtech-IK4 was partially financed by Colciencias (Colombian Administration for Science, Technology and Innovation).

References

AAFTAB MUNSHI, J. L. 2010. *OpenGL ES 2.0 Specification*. Khronos OpenGL ES Working Group.

AKENINE-MOLLER, T., AND HAINES, E. 2002. Real-time rendering.

BEHR, J., JUNG, Y., KEIL, J., DREVENSEK, T., ZOELLNER, M., ESCHLER, P., AND FELLNER, D. 2010. A scalable architecture for the HTML5/X3D integration model X3DOM. In *Proceedings of the 15th International Conference on Web 3D Technology*, ACM, 185–194.

BEHR, J., JUNG, Y., DREVENSEK, T., AND ADERHOLD, A. 2011. Dynamic and interactive aspects of X3DOM. In *Proceedings of the 16th International Conference on 3D Web Technology. Web3D*, vol. 11, 81–87.

DI BENEDETTO, M., PONCHIO, F., GANOVELLI, F., AND SCOPIGNO, R. 2010. SpiderGL: a JavaScript 3D graphics library for next-generation WWW. In *Proceedings of the 15th International Conference on Web 3D Technology*, ACM, vol. 1, 165–174.

HUMPHREYS, G., ELDRIDGE, M., BUCK, I., STOLL, G., EVERETT, M., AND HANRAHAN, P. 2001. WireGL: a scalable graphics system for clusters. In *Proceedings of the 28th annual conference on Computer Graphics and interactive techniques*, ACM, 129–140.

MAGALLÓN, M., HOPF, M., AND ERTL, T. 2001. Parallel volume rendering using PC graphics hardware. In *Computer Graphics and Applications, 2001. Proceedings. Ninth Pacific Conference on*, IEEE, 384–389.

MARRIN, C. 2011. *WebGL Specification*. Khronos WebGL Working Group.

MORENO, A., SEGURA, Á., KORCHI, A., POSADA, J., AND OTAEGUI, O. 2011. Interactive urban and forest fire simulation with extinguishment support. *Advances in 3D Geo-Information Sciences*, 131–148.

NEAL, B., HUNKIN, P., AND MCGREGOR, A. 2011. Distributed OpenGL rendering in network bandwidth constrained environments.

SHI, S., JEON, W. J., NAHRSTEDT, K., AND CAMPBELL, R. H. 2009. Real-time remote rendering of 3D video for mobile devices. In *Proceedings of the seventeen ACM international conference on Multimedia - MM '09*, ACM Press, New York, New York, USA, 391.

SONS, K., KLEIN, F., RUBINSTEIN, D., BYELOZYOROV, S., AND SLUSALLEK, P. 2010. XML3D: Interactive 3D graphics for the Web. In *Proceedings of the 15th International Conference on Web 3D Technology*, ACM, New York, NY, USA, Web3D '10, 175–184.

TIZON, N., MORENO, C., CERNEA, M., AND PREDA, M. 2011. MPEG-4-based adaptive remote rendering for video games. In *Proceedings of the 16th International Conference on 3D Web Technology*, ACM, vol. 1, 45–50.

ULBRICH, C., AND LEHMANN, C. 2012. A DCC pipeline for native 3D graphics in browsers. In *Proceedings of the 17th International Conference on 3D Web Technology*, ACM, 175–178.

ZAKAI, A. 2011. Emscripten: An LLVM-to-Javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, ACM, 301–312.