SPIDERGL: A GRAPHICS LIBRARY FOR 3D WEB APPLICATIONS

M. Di Benedetto*, M. Corsini, R. Scopigno

Visual Computing Lab, ISTI - CNR

KEY WORDS: Web Applications, Web Graphics, WebGL, Virtual Museums, Real-Time Graphics, Rich Multimedia Content

ABSTRACT:

The recent introduction of the WebGL API for leveraging the power of 3D graphics accelerators within Web browsers opens the possibility to develop advanced graphics applications without the need for an ad-hoc plug-in. There are several contexts in which this new technology can be exploited to enhance user experience and data fruition, like e-commerce applications, games and, in particular, Cultural Heritage. In fact, it is now possible to use the Web platform to present a virtual reconstruction hypothesis of ancient pasts, to show detailed 3D models of artefacts of interests to a wide public, and to create virtual museums.
We introduce SpiderGL, a JavaScript library for developing 3D graphics Web applications. SpiderGL provides data structures and algorithms to ease the use of WebGL, to define and manipulate shapes, to import 3D models in various formats, and to handle asynchronous data loading. We show the potential of this novel library with a number of demo applications and give details about its future uses in the context of Cultural Heritage applications.

1. INTRODUCTION

1.1 Computer Graphics and the World Wide Web

The delivery of 3D content via the Web platform started to be a topic of interest since the graphics hardware of commodity personal computers became enough powerful to handle non-trivial 3D scenes in real-time. Many attempts have been done to allow the user of standard Web documents to directly access and interact with three-dimensional objects or, more generally, complex environments from within the Web browser. Historically, these solutions were based on software components in the form of proprietary and often non-portable browser plug-ins. The lack of a standardized API did not allowed Web and Computer Graphics (CG) developers to rely on a solid and widespread platform, thus losing the actual benefits that these technologies could provide.
In the same period of time in which GPUs showed a tremendous increase in performances and capabilities, the evolution of the technology behind Web browsers allowed interpreted languages such as JavaScript to perform quite efficiently in general purpose computations, thanks to novel just-in-time (JIT) compilers. Thus, on one side, the hardware and software components have reached a level of efficiency and performances which could fit the requirements for high-quality and interactive rendering of 3D content to be visualized, on the other the increase of bandwidth for accessing the Internet allowed large volumes of data to be transferred worldwide in a relatively short amount of time.
In this scenario, the need for a standardized computer graphics API became a high-priority problem to be solved. In fact, in late 2009, the Khronos Group (KHRONOS GROUP, 2009) officialised a new standard, WebGL (KHRONOS-WEBGL, 2009), which aims at harnessing the power of graphics hardware directly within Web pages through a JavaScript interface. WebGL is an API specification designed to closely match the OpenGL|ES 2.0 specifications (KHRONOS-OPENGLES, 2009), with some modifications which make the API more close in look-and-feel to a JavaScript developer. On the other side, as web pages which use WebGL are freely accessible from every potential Web client, the new specification impose a series of restrictions to comply with a more strict security policy.
Although this scripting language cannot be considered as performing as a compiled one like C++, the tendency of delegating the most time-consuming parts of a CG algorithm to the graphics hardware helps mitigating the performance gap.

1.2 Leveraging the new Web Technologies

Thanks to the combination of hardware and software capabilities and performances, coupled with a high-speed data channel, it is nowadays possible to effectively and natively handle real-time 3D graphics within Web pages. In particular, by exploiting the asynchronous features provided by the runtime environment of the Web browser, it is possible to manage large datasets in a natural out-of-core fashion. The creation of fast and reliable visualization algorithms that allow the user to explore huge environments (like Google Earth (Google Inc., 2010) and Bing Maps (Microsoft Corporation, 2010)) implies that multiresolution algorithms should be developed with network streaming in mind, both in terms of caching mechanisms and the actual representation of a data packet. Alongside, it is easy to see how the new WebGL 3D technology will bring closer web developers, which are more and more interested in learning 3D graphics and CG developers, which will try to deploy their algorithms to less powerful platforms.

The question is now what still separates a compiled C++ from a JavaScript application with respect to CG algorithms. One obvious answer is execution speed, but there are other gaps to be filled:
- **Asynchronous content loading:** many CG algorithms, especially when dealing with multiresolution datasets, make intensive use of multithreading for asynchronous (down)loading of textures or geometry data from different cache levels. This is necessary to avoid the application to freeze while waiting for a texture to be loaded from RAM, disk or even a remote database to GPU. On the other hand JavaScript still does not officially support multithreaded execution.

- Shape data loading from file: there are many file formats for 3D models and as many C++ libraries to load them (CGAL Project, Visual Computing Lab, RTWH). JavaScript includes a series of predefined types of objects for which the standard language bindings expose native loading facilities (i.e. the Image object), but such bindings for 3D models have yet to come.
- Math: linear algebra algorithms for 3D points and vectors are very common tools for the CG developer, and a large set of dedicated libraries exists for C++ and other languages. Although many JavaScript demos for mathematical algorithms can be found just browsing the Web, a structured library with the specific set of operations used in CG is still missing.
- WebGL wrapping: the WebGL specification is very similar to OpenGL|ES 2.0, which means that there are significant changes with respect to OpenGL are, for example there are no matrix or attribute stacks and there is no *immediate mode*. Although these choices comply with the bare-bones philosophy of OpenGL|ES 2.0, they also imply incompatibility even with OpenGL 3.0, which, for example, still provides matrix stack operations.

In this paper we present *SpiderGL*, a JavaScript library designed to fill these gaps: it extends JavaScript by including geometric data structures and algorithms and wraps their implementation towards WebGL. In particular, SpiderGL was designed keeping in mind three fundamental qualities:
- Efficiency: with JavaScript and WebGL, efficiency is not only a matter of asymptotic bounds on the algorithms, but the ability to find the most efficient mechanism to implement, for example, asynchronous loading or parameters passing to the shader programs, without burdening the CPU with respect to a bare bone implementation;
- Simplicity and Short Learning Time: users should be able to reuse as much as possible of their former knowledge on the subject and take advantage of the library quickly. For this reason SpiderGL carefully avoids over-abstraction: almost all of the function names in SpiderGL have a one to one correspondence with either OpenGL or GLU commands (e.g. the SpiderGL function sglLookAt for setting up the camera pose matrix), or with geometric/mathematics entities (e.g. SglSphere3, SglMeshJS).
- Flexibility: SpiderGL does not try to hide native WebGL functions; instead it provides higher level functionalities that fulfil the most common needs of the CG developer, who can use SpiderGL and WebGL calls almost seamlessly.

2. RELATED WORK

2.1 3D Graphics and the Web

The delivery of 3D content through the WWW comes with a considerable delay with respect to other digital media such as text, still images, videos and sound. Just like it already happened for commodity platforms, 3D Computer Graphics is the latest of the abilities acquired by the Web browsers. The main reason for this delay is likely the higher requirements for 3D graphics in terms of computational power. In the following we summarize the technologies that have been developed over the years.
The *Virtual Reality Modeling Language* (VRML) (Ragget, 1995) (then superseded by *X3D* (Brutzmann and Daly, 2007)) was proposed as a text based format for specifying 3D scenes in terms of geometry and material properties, while for the rendering in the Web browser it is required the installation of a platform specific plug-in.
Java Applets are probably the most practiced method to add custom software components, not necessarily 3D, in a Web browser. The philosophy of Java applets is that the URL to the applet and its data are put in the HTML page and then executed by a third part component, the Java Virtual Machine (JVM). The implementation of the JVM on all the operating systems made Java applets ubiquitous and the introduction of binding to OpenGL such as JOGL (JOGL) added control on the 3D graphics hardware.
A similar idea lies behind the ActiveX (ACTIVEX) technology, developed by Microsoft since 1996. Unlike Java Applets, ActiveX controls are not byte code but dynamic linked Windows libraries which share the same memory space as the calling process (i.e. the browser), and so they are much faster to execute.
These technologies allow incorporating 3D graphics in a Web page but they all do it by handling a special element of the page itself with a third party component. More recently, Google started the development of a 3D graphics engine named O3D (Google Labs, 2009). O3D is also deployed as a plug-in for browsers, but instead of a black-box, non programmable control, it integrates into the browser itself, extending its JavaScript with 3D graphics capabilities relying both on OpenGL and DirectX. O3D is scene-graph-based and supplies utilities for loading 3D scenes in several commonly used formats.

2.2 WebGL Libraries

WebGL (KHRONOS-WEBGL, 2009) is an API specification produced by the Khronos Group (KHRONOS GROUP, 2009) and, as the name suggests, defines the JavaScript analogous of the OpenGL API for C++. WebGL closely matches OpenGL|ES 2.0 and, most important, uses GLSL as the language for shader programs, which means that the shader core of existent applications can be reused for their JavaScript/WebGL version. Since WebGL is a specification, it is up to the web browsers developer to implement it. At the time of this writing it is supported in the most used web browsers (Firefox, Chrome, Safari), and a number of JavaScript libraries are being developed to provide higher level functionalities to create 3D graphics applications.
For example WebGLU (Benjamin DeLillo, 2009) provides wrappings for placing the camera in the scene or for creating simple geometric primitives, other libraries such as GLGE (Paul Brunt, 2010) or SceneJS (Lindsay Kay, 2009) uses WebGL for implementing a scene graph based rendering and animation engines.

3. THE SPIDERGL GRAPHICS LIBRARY

Most of the current JavaScript graphics libraries implement the scene graph paradigm. Although scene graphs can naturally represent the idea of a “scene”, they also force the user to resort to complex schemes whenever more control over the execution flow is needed. There are several situations in which fixed functionalities implemented by scene graph nodes cannot be easily combined to accomplish the desired output, thus requiring the developer to alter the standard behaviour, typically by deriving native classes and overriding their methods or, in some cases, by implementing new node types. In these cases, a procedural paradigm often represents a more practical choice. Also, scene graphs contain a large codebase to overcome the

limitations of strongly typed imperative programming languages, which is no more required in dynamic languages such as JavaScript.

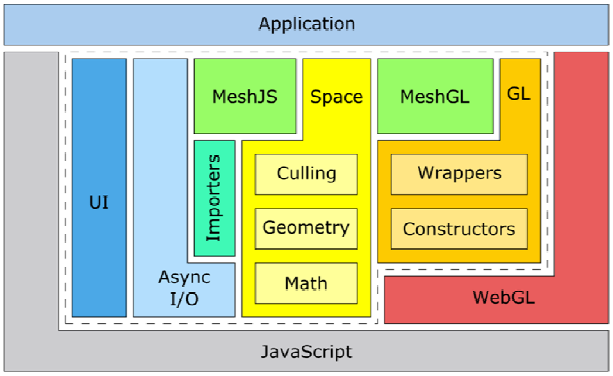

Figure 1. SpiderGL Library Architecture.

3.1 Library Architecture

SpiderGL is composed of the following five modules, distinguished by different colours in Figure 1:

- MATH: Math and Geometry utilities. Linear algebra objects and functions, as well as geometric entities represent the base tools for a CG programmer.

- GL: Access to WebGL functionalities. The GL module contains a low-level layer, managing low-level data structures with no associated logic, and a high-level layer, composed of *wrapper* objects, plus a series of orthogonal facilities.

- MESH: 3D model definition and rendering. This module provides the implementation of a polygonal mesh (SglMeshJS), to allow the user to build and edit 3D models, and its image on the GPU side (SglMeshGL). SpiderGL handles the construction of a SglMeshGL object from a SglMeshJS.

- ASYNC: Asynchronous Content Loading. Request objects, priority queues and transfer notifiers help the programmer to implement the asynchronous loading of data.

- UI: User Interface. A GLUT-like framework and a series of typical 3D manipulators allows a quick and easy setup of the web page with 3D viewports and provide effective management of user input.

By combining the core functionalities of each module, SpiderGL offers a series of practical and efficient solutions to implement the most common graphics tasks, described in the following.

Space-Related Structures and Algorithms
An important part at the foundation of a 3D graphics library comprises standard geometric objects, as well as space-related algorithms. SpiderGL offers a series of classes representing such kind of objects, like rays for intersection testing, infinite planes, spheres and axis aligned boxes, coupled with distance calculation and intersection tests routines.

Hierarchical Frustum Culling
When operating over a network, it is reasonable to assume that the content retrieval has a consistent impact on the overall performance. Since multimedia context began to be widely used in web documents, it was clear that a sort of multiresolution approach should have to be implemented to compensate for the transmission lags, giving the user a quick feedback, even if at a lower resolution (i.e. progressive JPEG and PNG). Following this principle, geometric *Level-Of-Detail* (LOD) techniques are used to implement a hierarchical description of a three-dimensional scene, where coarse resolution data is stored in the highest nodes of a tree-like structure while full resolution representation is available at the leaf level. To ease the use of hierarchical multiresolution datasets, SpiderGL provides a special class, SglFrustum, which contains a series of methods for speeding up the visibility culling process and projected error calculation for hierarchical bounding volumes hierarchies.

Matrix Stack
Users of pre-programmable (*fixed pipeline*) graphics libraries relied on the so called *transformation matrix stacks* for a logical separation among the projection, viewing and modelling transformations, and for a natural implementation of hierarchical relationships in composite objects through matrix composition.
Even if this has proven a widely used pattern, it no longer exists since version 2.0 of OpenGL|ES (it was claimed that its introduction in the specifications would have violated the principle of a bare-bones API). We thought that this important component was indeed essential in 3D graphics, so we introduced the SglMatrixStack class, which keeps track of a stack of 4x4 transformation matrices with the same functionalities of the OpenGL matrix stack. Moreover, the SglTransformStack comprises three matrix stacks (projection, viewing and modelling) and represents the whole transformation chain, offering utility methods to compute viewer position, viewing direction, viewport projection of model coordinates to screen coordinates and the symmetric unprojection. Note that, for practicality of use, we decided to have the modelling and viewing transformation stacks separated, contrarily to the single OpenGL *ModelView* stack.

3.2 Managing 3D Meshes

One of the fundamental parts of a graphics library consists of data structures for the definition of 3D objects (meshes) and their rendering. As in many libraries for polygonal meshes, SpiderGL encodes a mesh as a set of vertices and connectivity information. Following the philosophy of WebGL, a vertex can be seen as a bundle of data, storing several kind of quantities such as geometric (position, surface normal), optical (material albedo, specularity) or even *custom* attributes. The connectivity describes how these vertices should be connected to form geometric primitives, such as line segments or triangles.
As the representation of meshes is tightly related to their intended use, SpiderGL supplies two different data structures: the first one, SglMeshJS, resides in *client scope*, i.e. in system memory, where it can be freely accessed and modified within the user script; the other is SglMeshGL, which is the image of a mesh in the GPU memory under the form of WebGL vertex and index buffer objects. Crucial for memory and execution efficiency is how the vertex set and the connectivity information is laid out; in the following paragraphs we will describe our solution, mainly dictated by the JavaScript language and the WebGL execution model.

Vertices Memory Layout
There are two main data layouts which can be used to store vertex data: *array-of-structs* or *struct-of-arrays*.

In the first case, a vertex is represented as an object containing all the needed attributes: the vertex storage thus consists of an array of such vertex objects. In the second case, an array is created for each vertex attribute: in this case the vertex storage is a single object whose fields are arrays of attributes, where a vertex object is extracted by selecting corresponding entry in each array.
SpiderGL adopts the struct-of-arrays layout for two reasons: first, JavaScript runtime performs more efficiently when working with homogeneous arrays of numbers rather than arrays of generic object references; second, adding and removing attributes is easily accomplished. In a similar way, the GPU-side mesh (SglMeshGL) stores its vertices with a dedicated *vertex buffer object* (VBO) for each attribute.

Connectivity Memory Layout
The connectivity can be implicitly derived from the order in which vertices are stored or, more frequently, explicitly described with a set of vertex indices. In SpiderGL it is possible to represent both of them with, respectively, *array primitive* or *indexed primitive streams*.
A SpiderGL mesh may contain more than one primitive stream; for example it may contain a primitive stream for the triangles and one for the edges in order to render the object in a filled or wireframe mode; the main reason behind this choice is that OpenGL|ES 2.0 (and thus WebGL) specifications does not contain any routine to select the mode in which source geometric primitives should be rasterized. For example, such a routine (known in desktop OpenGL as glPolygonMode) could be able to setup the rasterizer in order to draw just the edges of a triangle primitive.

Overcoming WebGL Limitations
When using indexed primitives in WebGL, the native type for the elements in the index array can be either a 8- or a 16-bit unsigned integer; thus, the maximum addressable vertex has index 65536. SpiderGL automatically overcomes this limitation by splitting the original mesh into smaller sub--meshes. In order not to burden the user with special cases when converting an SglMeshJS to its renderable representation, we introduced the *packed-indexed primitive stream* for SglMeshGL, which transparently keeps track of sub-meshes bounds without introducing additional vertex or index buffers.

Rendering
In WebGL the rendering process involves the use of shader programs, vertex buffers and, often, index buffers and textures. Central to the graphics pipeline is the concept of binding points, that is, named input sites to which resources are attached and from which pipeline stages fetch data. To ease the connection between mesh attributes and shader attributes, and to provide an efficient rendering process, SpiderGL provides the SglMeshGLRenderer class. It ensures that the minimum amount of work is demanded to the WebGL implementation, while exhibiting a simple interface even for complex tasks.

4. USING SPIDERGL

At the time of writing, the WebGL specification is in its final draft version and it is implemented in the experimental version of major web browsers. We successfully tested our library with the latest builds of the most common web browsers on several desktop systems.
The results presented here have been run on the Chromium web browser on a Windows Vista system with Intel i7 920 processor, 3 GB RAM, 500 GB Hard Drive and an NVIDIA GT260 graphics board with screen vertical synchronization disabled. The collected results should be analyzed by considering that a minimal HTML/JS page that only clears the color buffer reaches the limit of exactly 250 frames per seconds; we suspect that some kind of temporal quantization occurs in the browser event loop.

4.1 Standard CG Algorithms and Data Issues

As noted in Section 2.2, WebGL can be considered as a one-to-one mapping of OpenGL|ES 2.0 functions to a JavaScript API: this means that not all the functionality in even not-so-recent versions of standard OpenGL is available to the developer.

Shadow Mapping: the first example consists of rendering a 100K triangles mesh of a 3D-scanned artefact, using Phong lighting model and a 1024x1024 shadow map (see Figure 2), which can be done at full framerate (250 FPS). It can be noted how the use of projected shadows, as well as self shadows, can greatly enhance the perception of spatial relationships, both at the scene level (among different objects) and at the object level (model features).

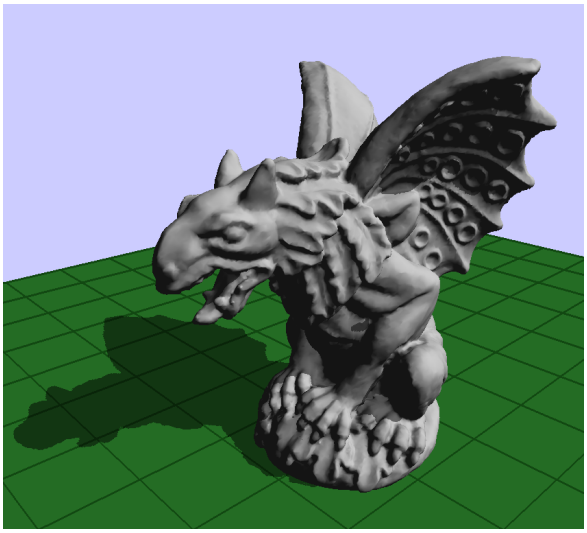

Figure 2. Shadow mapping algorithm enhances space perception. Here it is shown a 100K triangles mesh obtained from a 3D scan of a real artefact.

Large Meshes: when dealing with virtual replicas of real objects, among the most valuable attributes of interest for the Cultural Heritage is the ability to be as close as possible to the real measurements. That is, a large amount of geometric data is needed to represent the virtual 3D object. From the point of view of a WebGL visualization application, this translates in the need for handling an amount of vertices that is far beyond the system capabilities.
To highlight the capabilities of the *packed-indexed primitive stream* (see Section 3.2), Figure 3 shows a 3D scan of Michelangelo's David statue composed of 1M triangles.

Caching: the general data flow and cache hierarchy used in compiled remote applications (remote online repository, disk, system RAM and video RAM) cannot be explicitly implemented in web application due to limitations imposed by the restrictive permissions policy adopted for security reasons by web browsers; for example, it is not possible to create and write files in the local file system. This means that we can not

explicitly implement the disk cache stage. In reality, even if we will not have explicit control over disk usage and cache eviction policy, by using standard Image and XmlHttpRequest objects we will automatically take advantage of the caching mechanisms implemented by the browser itself. In fact, every standard web browser caches recently transferred data in the local file system (and even system RAM), thus transparently providing a disk cache stage.

Figure 3. A zoomed view of the rendering of Michelangelo's David statue.

The model outreaches the maximum value for vertex indices and is thus automatically subdivided into smaller chunks, highlighted by different colors (see Figure 4).
The performances here range very inconstantly from 90 to 140 FPS, with peaks of 250. This is probably due to the way the timer event is scheduled by the browser.

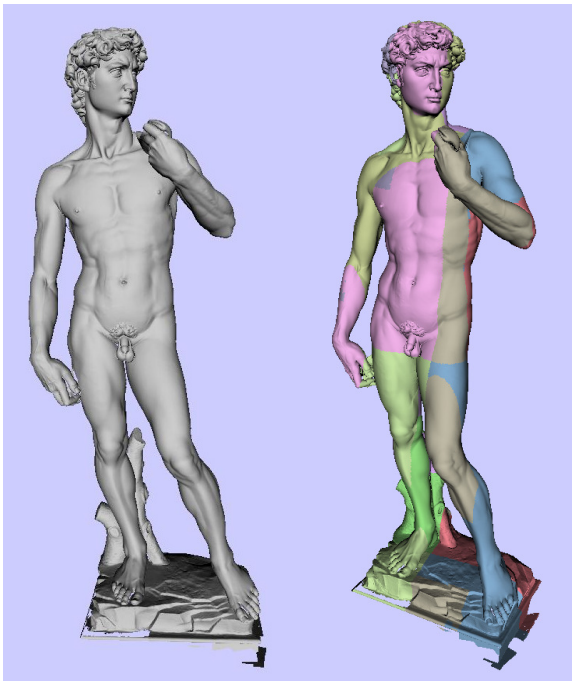

Figure 4. The 1M triangles model is automatically subdivided into sub-chunks (highlighted by color) to be rendered with WebGL.

4.2 Large Datasets on the Web

The large availability of geometric and color data, as well as network connections able to transfer up to several megabytes of data per second, impose a rendering library to be designed to maximize the performances in those areas which are typical of multiresolution rendering algorithms. The need for a multiresolution approach comes into play whenever we want to show datasets that are too large to be handled with respect to the hardware capabilities: in particular, the resources which mainly influence the output of a multiresolution renderer are System and Video RAM speed and amount, as well as the raw CPU and GPU performances.

Terrain Models
To show how the algorithms and data structures in SpiderGL can be easily used to integrate virtual 3D exploration inside web pages, we implemented a simple but effective multiresolution terrain viewer (see Figure 5). Our approach consists of an offline pre-processing step which creates a multiresolution representation from a discrete elevation and color image, and an online out-of-core rendering algorithm.
As in other existing map-based web applications like Google Maps (Google Inc., 2010), the multiresolution dataset is organized in tiles. More in detail, in the construction phase, the elevation map is first embedded on a quad-tree with user-defined depth. The depth of the tree determines the dimension of the tiles in which the input map is first partitioned. In fact, such tiles correspond to the 2D projection of the bounding box of the leaf nodes.
We build the multiresolution dataset in a bottom-up fashion by first assigning the tile images to their leaf nodes. Then, tiles for internal nodes are generated by assembling the four tiles assigned to the node children in a single square image of twice the dimensions and then down-sampled by a factor of two. This means that all the tiles have the same dimension and, in particular, tiles assigned to nodes at level i have half the linear resolution of the ones at level i+1. To ensure that the border of adjacent tiles match exactly to avoid cracks and discontinuities, the square regions of the elevation tiles are expanded by one pixel on each side. This is also done for the input color map to allow correct color interpolation at borders when bilinear filtering is used. The output of the preprocessing step are two texture images for each node: a RGBA image stores the surface color (RGB channels) and the height map (Alpha channel), and a RGB image for normal maps.
Once the nodes to be rendered are identified by the multiresolution algorithm, the rendering process draws each tile by using a vertex shader which performs displacement mapping on a regular triangle grid.

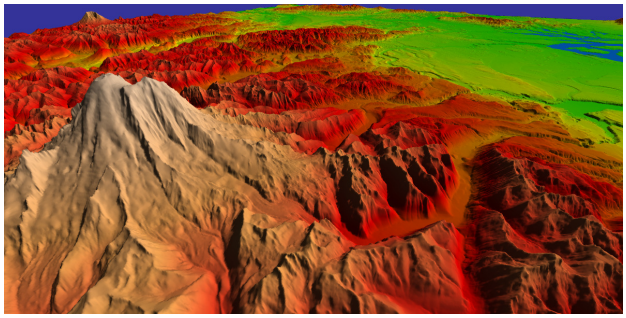

Figure 5. Remote multiresolution visualization of a large terrain model.

Urban Models

To show the real potential of WebGL, we implemented a multiresolution renderer of urban environments using a special data structure called *BlockMap* (Di Benedetto et al., 2009). One of the main advantages of this novel representation is that it can be directly encoded into plain 32-bit RGBA images, implying that we can use the standard tools natively provided by JavaScript (namely, the Image, Canvas and WebGLContext objects) to fetch data from remote repositories and upload it to the graphics hardware. Another advantage is that the simplicity of the rendering algorithm and, more importantly, the use of simple instructions in the BlockMap shaders, allow us to write efficient JavaScript code and exploit the actual power of WebGL without any modification.

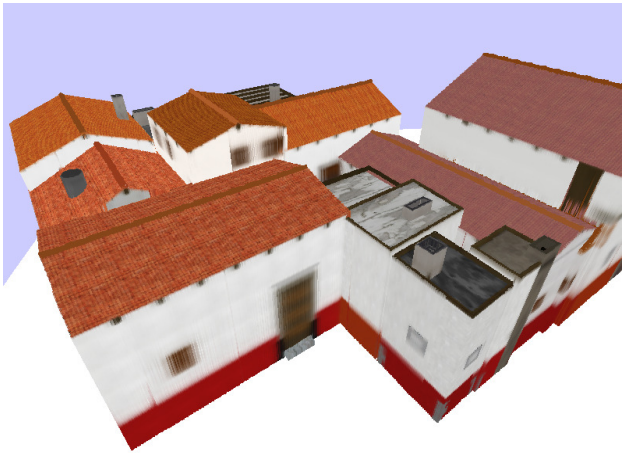

Figure 6. The Ray-Casted BlockMap technique is implemented in SpiderGL for real-time exploration of urban models.

5. VIRTUAL MUSEUMS

The possibility to use the processing capabilities of modern graphics hardware directly within web pages allows the development of new kind of virtual exploration software that can be seamlessly integrated in existing remote digital libraries. In particular, in conjunction with the growth in the availability of large 3D-scanned models and high resolution photographs, the new WebGL standard can be used to create interactive, high-quality virtual museums that can be accessed through the Web.

In the following we will describe two case-study that show how the exploration and rendering modules of a virtual museum can be efficiently implemented.

5.1 Relightable Images with Polynomial Texture Maps

Although WebGL is designed primarily for three-dimensional graphics, the possibility to use the power of the graphics hardware at pixel processing level makes it an attractive candidate even for complex 2D shading operations. In this case, source images are handled under the form of texture maps, and per--pixel operations are executed by fragment shaders. As an effective use of these capabilities, we used the multiresolution framework to implement a *Polynomial Texture Map* (PTM) viewer. A Polynomial Texture Map (Malzbender, 2004; Dellepiane et al. 2006) is, in brief, a discrete image where each pixel encodes a minimal reflection function which depends on the light direction. This allows the user to interactively relight the image to facilitate visual inspection of fine details. To display such kind of image remotely, as in a streaming terrain viewer, a quad-tree is built from the original PTM and, at rendering time, a fragment shader computes the current colour as a function of the light position, passed as a global uniform variable.

In the example we developed, a large PTM image (2930x2224 pixels, for a total of 56 Mbytes), shown in Figure 7, is progressively streamed and refined according to the zoom level. The user can change the position of the virtual light source by simply moving the mouse cursor. At each movement, the illumination contribution is recomputed in real-time and the relighted image is shown.

Figure 7. A Polynomial Texture Map is visualized in a multi-resolution fashion. As the user moves the mouse to change the virtual light source direction, the illumination contribution is recalculated and shown in real-time.

This technique is mainly used for object that exhibits a main 2D structure, such as bas-relief. The net effect is that the virtual exploration provides a more immersive experience, allowing to examine the objects under different lighting conditions.

5.2 MeShade: a 3D Content Authoring Tool

While there are very large repositories for pictures, video or audio files, a web site like Flickr or YouTube for 3D models has yet to come. Up to now there are a few repositories of 3D models made by human modellers that one can browse and also few examples of repository of 3D scanned models (Stanford Computer Graphics Laboratory, 2004; Falcidieno, 2004).

However it is likely that this will change quickly in the near future, both for the increasingly ease of producing 3D models by automatic reconstruction means (for example by cheaper and cheaper laser scanners (NEXTENGINE, 2010) or by digital photography (Vergauwen and Van Gool, 2006)) and for the ability to use 3D graphics hardware acceleration in the Web browser.

MeShade is a Web application written with SpiderGL that allows the user to load a 3D model and images, create a custom shader program (like one can do using, for example, AMD RenderMonkey (AMD, 2010), although at the present with a more limited number of functionalities), and export JSON and HTML code snippets to create a web page which will provide interactive visualization of the mesh using the custom shader.

The user interface of MeShade consists of several collapsible and movable panels (see Figure 8), representing the most important parts of a shader composer application.

Apart from the interactive preview viewport which displays the loaded 3D model with the current material, the user is provided with text areas for editing the source code of the vertex and the fragment shaders.

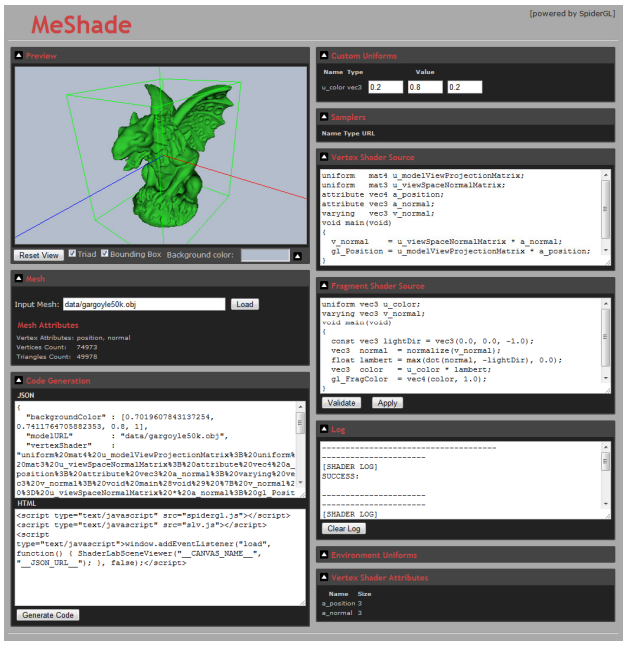

Figure 8. The MeShade user interface allows for the online editing of the material shader source code. When finished, a JSON script is generated to be included in remote repositories.

The user can validate the correctness of the shaders by using the *Validate* button which will show the compiler output (warning and error messages) in the log area. The *Apply* button will apply the shader program to the 3D model.
The way MeShade handles program uniforms and vertex shader attributes is based on predefined names with specific semantic and user-defined input values. In particular:
- every vertex attribute of the mesh is made available to the vertex shader by declaring it with a predefined prefix, i.e. vertex shader attribute *a_position* will be mapped to mesh vertex attribute stream named *position*;
- a series of fundamental and commodity values are exposed via predefined uniform names, like transformation matrices, model bounding box and so on;
- whenever a non-predefined uniform is found, an edit form is added to the HTML DOM which allows for direct editing of the scalar or vector values; the user interface for editing depends on the type of the uniform variable;
- an image load form is created for every texture sampler uniform; although texture samplers are standard uniforms in the GLSL language, they are grouped in a separate panel to reflect the way SpiderGL handles textures.

The 3D model and the texture images are loaded by specifying their URL and then pressing the corresponding *Load* button. The interface also contains a list of all available predefined uniforms and mesh vertex attributes. The latter ones are updated every time a model is loaded.
Once the user has reached a satisfactory result, he/she can ask MeShade to generate the code to embed the 3D model rendered with the program shader just created within a web page, by pressing the *Generate Code* button. MeShade will generate two code fragments, JSON and HTML, which can be copied to new or existing files.

The JSON section contains the geometry and images locations, as well as the shaders source code and uniform values, and thus serves as a *scene* description file. On the other side, the HTML code contains all the HTML *script* tags to be pasted into existing pages in order to access and visualize the scene.
We decided to generate code snippets rather than a complete HTML page because repository designers are supposed to use their own graphical style throughout their web sites: having only a very few lines of code to embed inside web pages allows for a variety of design choices. Moreover, separating the JSON scene description code allows for sharing the same scene among several web pages without code replication.

6. CONCLUSIONS

The work presented in this paper shows the potentiality of a WebGL-based application and the new possibilities opened by the availability of the modern graphics hardware features within the Web browsers. The proposed SpiderGL library made easy the coding of complex Computer Graphics applications by giving the developer the necessary tools for mathematical entities, 3D models management, data retrieval, and, efficient and fully-configurable rendering mechanisms. Exploiting these new Web technologies in the field of Cultural Heritage will allow, in the near future, to create remote virtual museums able to provide immersive and detailed exploration applications not confined to small objects but including very detailed artefact and even large environments like reconstructions of ancient cities.

Future Work

Beside the work of upgrading and extending SpiderGL, which is obviously a daily activity, we envisage a promising direction of work in the automatization of the process of converting large databases of scanned objects to Web repositories. The problems are mainly related to the typical large size of scanned objects and to the way to optimize them for a remote visualization. Although there are many available tools to reduce the number of polygons in a mesh, to parameterize it and to recover almost the full detail by bump mapping techniques (just to mention a viable, not unique, optimization pipeline), the whole process still requires a skilled user to be done.

REFERENCES

ACTIVEX, *"Microsoft ActiveX Controls"*, http://msdn.microsoft.com/

AMD, "Render Monkey", 2010. http://ati.amd.com/developer/rendermonkey/

Bianca Falcidieno, *"AIM@SHAPE Project Presentation"*, IEEE Computer Society, pp. 329-338, 2004.

Paul Brunt, *"GLGE: WebGL for the lazy"*, 2010, http://www.glge.org/

Don Brutzmann and Leonard Daly, *"X3D: Extensible 3D Graphics for Web Authors"*, Morgan Kaufmann, 2007.

CGAL Project, *"CGAL: Computational Geometry Algorithms Library"*, http://www.cgal.org/

M. Dellepiane, M. Corsini, M. Callieri, R. Scopigno, *"High Quality PTM Acquisition: Reflection Transformation Imaging for Large Objects"*, 7th International Symposium on Virtual Reality, Archaeology and Cultural Heritage (VAST2006), pp. 179-186, 2006.

Benjamin DeLillo, *"WebGLU: A utility library for working with WebGL",* 2009, http://webglu.sourceforge.org/

M. Di Benedetto, P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton and R. Scopigno, *"Interactive Remote Exploration of Massive Cityscapes"*, The 10th International Symposium on Virtual Reality, Archaeology and Cultural Heritage VAST (2009), pp. 9-16, 2009.

Google Labs, "O3D", 2009. http://code.google.com/apis/o3d/

Google Inc. , "Google Maps", 2010. http://maps.google.com/

Google Inc., *"Google Earth"*,http://www.google.com/earth/index.html

JOGL, *"JOGL Java Binding for the OpenGL API",* http://kenai.com/projects/jogl/pages/Home/

KHRONOS GROUP, *"Khronos: Open Standards for Media Authoring and Acceleration",* 2009. http://www.khronos.org/

KHRONOS-WEBGL, *"WebGL – Opengl\ES 2.0 for the Web.",* 2009. http://www.khronos.org/webgl/

KHRONOS-OPENGLES, *"OpenGL\ES - The Standard for Embedded Accelerated 3D Graphics"*, 2009. http://www.khronos.org/opengles/

Lindsay Kay, "SceneJS", 2009. http://www.scenejs.com/

T. Malzbender, *"Enhancement of Shape Perception by Surface Reflectance Transformation"*, Vision, Modeling, and Visualization, 2004.

Microsoft Corporation , *"Bing Maps",* www.bing.com/maps

NEXTENGINE, *"NextEngine",* 2010. http://www.nextengine.com/

D. Raggett, *"Extending WWW to support Platform Independent Virtual Reality",* Technical Report, 1995.

RWTH, *"OpenMesh: Visualization and Computer Graphics Library"*, http://www.openmesh.org/

Stanford Computer Graphics Laboratory, *"Stanford Repository",* 2000. http://graphics.stanford.edu/data/3Dscanrep/

M. Vergauwen and L. Van Gool, *"Web-Based 3D Reconstruction Service"*, Machine Vision Applications, vol. 17, pp. 411-426, 2006.

Visual Computing Lab, *"VcgLib: Visualization and Computer Graphics Library",* http://vcg.sourceforge.net