# SpiderGL: A JavaScript 3D Graphics Library for Next-Generation WWW

Marco Di Benedetto* Federico Ponchio† Fabio Ganovelli‡ Roberto Scopigno§

Visual Computing Lab
ISTI-CNR

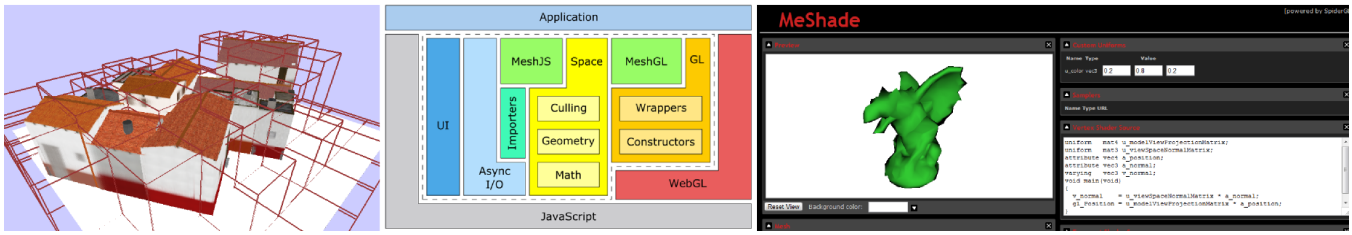



**Figure 1:** *Left: rendering a BlockMap with SpiderGL. Center: architecture of SpiderGL. Right: MeShade: a SpiderGL-based tool for shaders authoring.*

## Abstract

Thanks to the WebGL graphics API specification for the JavaScript programming language, the possibility of using the GPU capabilities in a web browser without the need for an ad-hoc plug-in is now coming true. This paper introduces 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, to handle asynchronous data loading. We show the potential of this novel library with a number of demo applications. Furthermore, we introduce MeShade, a SpiderGL-based web application for shader material editing from within the web browser, which produces all the code needed for embedding interactive 3D model visualization capabilities inside web pages and online repositories.

## 1 Introduction

The hardware and software technologies for delivering 3D content have been constantly improving over these latest years. First of all, current off-the-shelf GPU's are now powerful parallel computers which may carry out many of the computer graphics tasks, lightening the load of the CPU so that less performant, interpreted languages, such as JavaScript, are up to the job of executing the CPU side of the algorithms. Secondly, the execution of JavaScript has dramatically sped up in the latest generation of web browsers thanks to novel just-in-time compilers such as TraceMonkey [Mozilla 2010], V8 [Google Labs ] and SquirrelFish [Apple Corps. ]. Finally, API's such as O3D [Google Labs 2009] or, more recently, WebGL [Khronos Group 2009b] make the GPU controllable by JavaScript, thus providing the link between the web browser and the GPU.

*e-mail: marco.dibenedetto@isti.cnr.it
†e-mail: federico.ponchio@isti.cnr.it
‡e-mail: fabio.ganovellii@isti.cnr.it
§e-mail: roberto.scopigno@isti.cnr.it

WebGL in particular is now a reference point for 3D graphics, for it is a Khronos specification of JavaScript bindings to OpenGL and OpenGL|ES 2.0, and it is being included in the next release of almost all the main web browsers (Firefox, Chrome, Safari).

It is easy to see how these changes 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 make intensive use of multithreading for asynchronous (down)loading of textures or geometry data from different cache levels. This is vital 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 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 ; VCG ; RWTH ]. 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: WebGL specification is very similar to OpenGL|ES 2.0, which means that there are significant changes w.r.t. OpenGL, for example there are no matrix or attribute stacks and there is no immediate mode. Although these choices comply to 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.

SpiderGL is a JavaScript library designed to fill these gaps: it ex-

tends JavaScript by including geometric data structures and algorithms and wraps their implementation towards WebGL. 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. `sglLookAt`), 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 fulfill the most common needs of the CG developer, who can use SpiderGL and WebGL calls almost seamlessly.

The contribution of this paper is twofold:

- It introduces SpiderGL to the Computer Graphics and Web community
- It provides a number of working and publicly available applications developed with SpiderGL, among which MeShade: an online application for deploying 3D models on the Web with customizable material.

The rest of this paper is organized as follows: Section 2 briefly summarizes the state of the art on this subject while Section 3 illustrates the SpiderGL library, focusing on its organization and characteristics. Section 4 shows a few practical uses and applications developed with SpiderGL, while Section 5 details MeShade. Conclusions and future work conclude the paper in Section 6.

## 2 3D Graphics and the Web

The delivery of 3D content through the web 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) [Raggett 1995] (then superseded by X3D [Don Brutzmann 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. The implementation of the JVM on all the operating systems made Java applets ubiquitous and the introduction of binding to OpenGL such as JOGL [JOG ] added control on the 3D graphics hardware. A similar idea lies behind the ActiveX [ACT ] technology, developed by Microsoft since 1996. Unlike Java Applets, ActiveX controls are not bytecode 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 to incorporate 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.
WebGL [Khronos Group 2009b] is an API specification produced by the Khronos group [Khronos Group 2009a] 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 WebGL is supported in the nightly build versions of 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 [DeLillo 2009], which is the WebGL correspondent of GLU [Khronos Group ], provides wrappings for placing the camera in the scene or for creating simple geometric primitives, other libraries such as GLGE [Brunt 2010] or SceneJS [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 behavior, 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.

### 3.1 SpiderGL Architecture

SpiderGL is composed of five modules, distinguished by color in Figure 2:

- 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`.

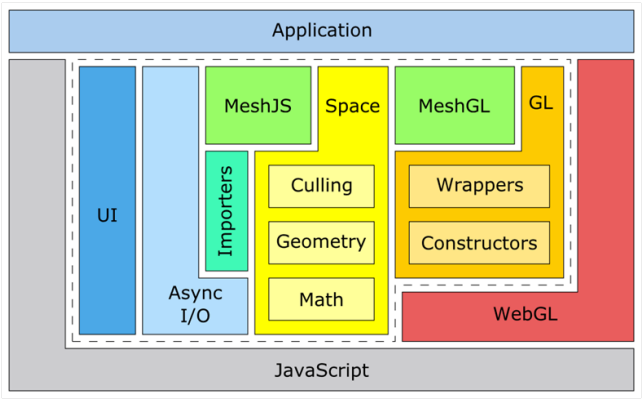



**Figure 2:** *A diagram of the SpiderGL architecture.*

- ASYNC : Asynchronous Content Loading.
- UI : User Interface.
- Space: Math and Geometry utilities.

### GL: Constructors

WebGL specifications expose an extremely low level API, according to the base philosophy of being a lightweight, highly configurable and high performances graphics infrastructure. However, a series of typical usage patterns can be extrapolated from most of the 3D graphics applications. As an example, the sequence of command for creating a shader program or framebuffer object are almost always the same block of code. The `GL` module exposes a number of easy-to-use creation functions that hide the most common operations and parameters to the developer.
However, since the user should be able to control all the low level details which are exposed by the WebGL standard, SpiderGL allows to override default parameters with *options* function parameters, in the philosophy of the JavaScript programming style. The general creation function would then be:

```
sglSomeGLObjectInfo(
    gl, arg1, /* ..., */ argN, options
);
```

where `gl` is the WebGL context object, `arg1..N` are mandatory object-specific parameters (like texture width and height), and `options` is a JavaScript object of the form

```
var options = {
    objSpecificParam1 : nonDefaultValue1,
    // ...
    objSpecificParamK : nonDefaultValueK
};
```

That is, whenever the user needs to override default values, a specific field in the `options` parameter can be specified with its respective value. This simple mechanism is indeed a powerful way to lessen the burden of library users in frequently performed tasks, while giving them the all the control they need whenever default parameters do not suffice.
For every type of WebGL object, the developer can use the corresponding SpiderGL creation function which returns a JavaScript object of type `Sgl`*`ObjectType`*`Info` with no associated logic (e.g. methods) where every field corresponds to an attribute of the native object. In the case of container objects like shader programs, WebGL functions are used to retrieve object specific values which are not part of the construction parameters set. This is the case of shader uniform locations and vertex attributes binding points. Whenever an object is created with these utility functions, the developer can freely use its `handle` field to directly work with WebGL calls.

**GL: Wrappers** The use of a low-level API often requires a sequence of calls even to accomplish a simple task and even after object/resource creation and initial setup. For this reason, every WebGL object has a corresponding higher-level wrapper which takes care of the usage details.

Wrapper objects constructors parameters are the same used in their corresponding lower-level creation functions, but they also have overloaded versions which take the single `Sgl`*`ObjectType`*`Info` structure. This is particularly useful when more developers are involved in a large system: in this case, everyone can choose the best level of abstraction which fits his or her habits. For example, if WebGL resources are globally accessible, the developer of a first module can decide to use the native `handle` reference, while in another module other people could choose a higher level approach by creating a wrapper object around the low level object definition, i.e.:

```
// wrap a shader program
var prgWrap = new SglProgram(gl, prgInfo);
```

and ensuring that every attribute of the object is properly set to meet the wrapper assumptions. To this end, every wrapper contains a `synchronize()` method which retrieve the salient attribute values which could have been changed with native WebGL calls. For performance reasons, the synchronization step is not automatically performed but should be explicitly invoked.

### MESH: Mesh Manipulation and Rendering

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 any library for polygonal meshes SpiderGL encodes a mesh as a set of vertices and connectivity information.
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 vertex/index buffer objects.

**Memory layout: vertices** There are two main 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:

- JavaScript runtime performs more efficiently when working with homogeneous arrays of numbers rather than arrays of generic object references

- 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. Such a choice for the GPU layout can be objected by claiming that the use of *interleaved vertex arrays* is more efficient, because interleaving attributes results in a more efficient memory access pattern, which is true when we solely consider the memory bandwidth used by the isolated system composed of the *GPU memory* and the *vertex puller* stage of the graphics pipeline. In this subsystem, the prefetching policy of the *pre-transform-and-lighting cache* mitigates the transfer latency.
However, when looking at the whole rendering pipeline in real-life scenarios, we see how the most relevant part of the execution time is spent in the vertex shader (whose overall performance is increased by the *post-transform-and-lighting cache*), in the fragment shader, in texture accesses and framebuffer writes or compositing (blending) operations. In our experiments, these bottlenecks made the benefits of the interleaved layout not even measurable.
These considerations supported our choice in adopting the *struct-of-arrays* layout for both Application-side and GPU-side meshes.

**Memory layout: connectivity** 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 (please note that the `glPolygonMode` command is not in the WebGL specifications).

**Overcoming WebGL limitations** When using indexed primitives in WebGL, the native type for the elements in the index array can be `UNSIGNED_BYTE` or `UNSIGNED_SHORT`, so the maximum vertex index is $2^8 - 1$ or $2^{16} - 1$, respectively. SpiderGL automatically overcomes this limitation by splitting the original mesh into smaller packets. 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 (see Figure 3).

Rendering a mesh with WebGL consists of the following steps:

1. Attach mesh vertex buffers to named vertex attributes binding sites
2. In case of indexed primitives, attach the index buffer to the primitive index binding site
3. Bind a shader program to the vertex and fragment processing stages
4. Establish a correspondence between vertex attribute binding points and vertex shader input attributes
5. Invoke the rendering command

In SpiderGL, mesh rendering is efficiently accomplished with the `SglMeshGLRenderer` helper class. This class takes care of all

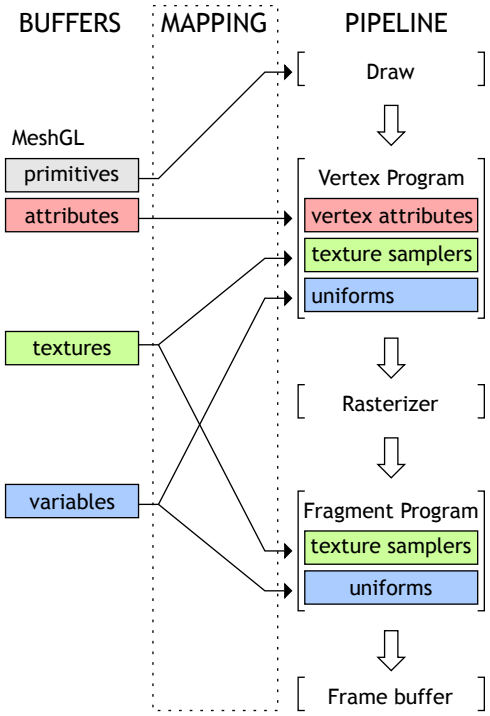



**Figure 3:** *Mesh Rendering: vertex buffers, index buffer and textures are attached to named binding sites. Then a series of correspondences is established between: a) mesh vertex streams and vertex shader attributes, b) texture units and shader texture samplers and c) application values and shader uniforms.*

the steps listed above, minimizing the number of binding operations in case of multiple instances rendering and providing smart mechanisms which allow the user to explicitly setup vertex attributes correspondences, shader program uniforms and texture samplers.

**Importers: Input Formats** Content formatting (geometry, images, shader effects code etc.) has been carefully kept away from OpenGL-class graphics libraries, due to their application-specific nature; therefore, the importers are not really part of the kernel of SpiderGL. At the present SpiderGL supports COLLADA, the Alias|Wavefront `OBJ` format for basic three-dimensional objects and, due to the extremely simple definition of mesh, the `JSON` [Crockford ] object serialization framework.

### 3.1.1 ASYNC: Asynchronous Content Loading

Several CG applications requires the ability of asynchronous loading in memory from the disk or from a remote location. Usually this is implemented using a multithread architecture with prioritized request queues which prevent the rendering freeze during data transfers. JavaScript does not support multithreading so the asynchronous loading of remote content is done by using `XMLHttpRequest` and `Image` objects, appropriately set up with callback functions which will be invoked whenever the transfer of the requested data has completed. SpiderGL uses this mechanism to implement prioritized request queues. The following code snippet shows how to use a priority queue to create textures from remote images:

```
// create a request queue
// with a maximum number of ongoing requests
var requestQ = new SglRequestQueue(maxOngoingReqCount);
```

```
// define a callback function that will
// create a texture when the image data is ready
var textures  = { };
var callback = function(request) {
    textures[request.url] = new SglTexture2D(gl, ↩
        request.image);
};

// instantiate requests and push it into the queue
var req1 = new SglImageRequest("sourceURL1", callback);
requestQ.push(req1); // continue issuing requests
```

The status of the request can be queried at any time, allowing the application to take different code paths depending on the resource availability or even abort the request. This whole mechanism is particularly useful when multiresolution algorithms and data structures are employed for tasks such as 3D navigation in large environments. Along with asynchronous facilities, this module provides easy-to-use routines to access the internal content of the HTML document.

## 3.2 UI: Event Handling and Interactors

To access the WebGL layer within a web page, a specific context object must be requested from an HTML `canvas` object. The HTML rendering engine will then issue a page composition operation whenever it detects changes to the associated WebGL framebuffer. In every interactive application, the displayed content is often a consequence of some kind of user interaction. SpiderGL provides an event handling subsystem which reflects the philosophy of the GLUT library [Kilgard ]. GLUT allows application developers to selectively install custom callbacks for the most common user input event types, such as keyboard key presses or mouse motion. We translated the callback approach used in GLUT in a more object oriented way. The developer can create a generic object and register it to an HTML `canvas` element; events raised from the canvas will be then intercepted by the user interface module which in turn will dispatch them to the registered object. The correspondences between events and event handlers are resolved by simply inspecting the registered object and looking for methods with predefined names. In addition to event dispatching, the registered object will be augmented by a `ui` field which can be used to access the logging system, as well as important information such as tracked input state and canvas properties.
Interaction with the three-dimensional scene itself is done with standard viewpoint and object manipulation tools. These include a camera interactor which implements the typical paradigm used in first-person shooter games (`SglFirstPersonCamera`) and a trackball manipulator (`SglTrackball`) for object inspection with pan, zoom, rotation and scaling operations.

## 3.3 GEOMETRY: Math and Space

The geometry module provides low-level mathematical functions and objects as well as space-related object representations and algorithms, as described in the following.

### 3.3.1 Math

This submodule implements essential mathematical objects such as vectors and matrices, along with basic operations on them. Particular attention has been paid in their implementation in order to reduce the programmer effort and the impact on performances.
The low-level layer is composed by functions that operate on native JavaScript arrays as input parameters and return types. As a usage example, calculating a three-dimensional triangle normal given 3 JavaScript arrays `v0,v1,v2` would be as follow:

```
var normal = sglNormalizeV3( sglCrossV3(sglSubV3(v1, v0↩
    ),sglSubV3(v2, v0)));
```

As it can be noted, a complex expression is expressed by nesting function calls. Moreover, the lack of overloading (which does not exist in JavaScript) imposes the use of unique names for distinguishing on the input types (in the above example, the `V3` suffix is used to identify the subset of functions working with 3-dimensional vectors).
The high-level layer is composed of classes that wrap the low-level layer. The above example would become:

```
var v0 = new SglVec3(x0, y0, z0);
var v1 = new SglVec3(x1, y1, z1);
var v2 = new SglVec3(x2, y2, z2);
var normal = v1.sub(v0).cross(
    v2.sub(v0)
).normalized();
```

The choice on which level of abstraction to use is up to the developer.

### 3.3.2 Space-Related Structures and Algorithms

Another important module 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 an 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) is 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 modeling 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 modeling) 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 we decided to have the modeling and viewing transforma-

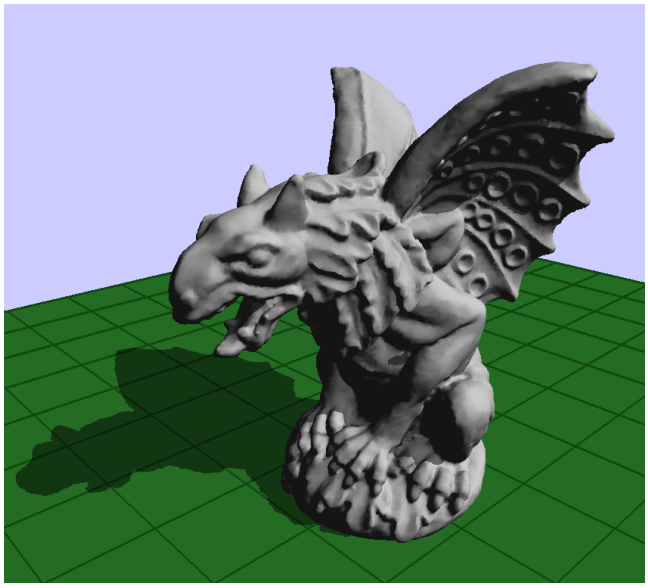


**Figure 4:** *Shadow Mapping: a scene composed of 100K triangles is rendered at the maximum reachable speed of 250 FPS. The* $1024^2$ *shadow map has been packed on a 32 bit RGBA texture because the browser does not support depth textures fetches in fragment programs.*

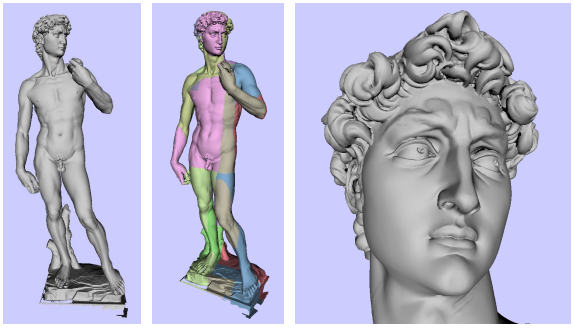
tion stacks separated, contrarily to the single OpenGL *modelview* stack.

## 4 Using SpiderGL

The WebGL specification is still in draft version and it is only implemented in the experimental version of almost all 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.

The first example consists of rendering a 100K triangles mesh using Phong lighting model and a $1024^2$ shadow map (see Figure 4), which can be done at full framerate (250 FPS). Note that the current WebGL specifications does not allow to read back values from depth textures inside a fragment shader, so in the shadow pass we encoded the fragment depth value in a 32 bit RGBA texture.

To highlight the capabilities of the *packed-indexed primitive stream* (see Section 3.1), Figure 5(a) shows a 3D scan of Michelangelo's David statue composed of 1M triangles. The model outreaches the maximum value for vertex indices ($2^{16} - 1$) and is thus automatically subdivided into smaller chunks (nine in this case), highlighted by different colors in Figure 5(b). 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.



(a) (b) (c)

**Figure 5:** *Large Meshes: vertex index limit is automatically overcome with* packed-indexed primitive stream. *Here a model of the Michelangelo's David statue with 1M triangles is rendered at about 100 FPS on a web browser. The figure shows: a) the whole mesh, b) the colored chunks after splitting and c) a close-up of the statue head.*

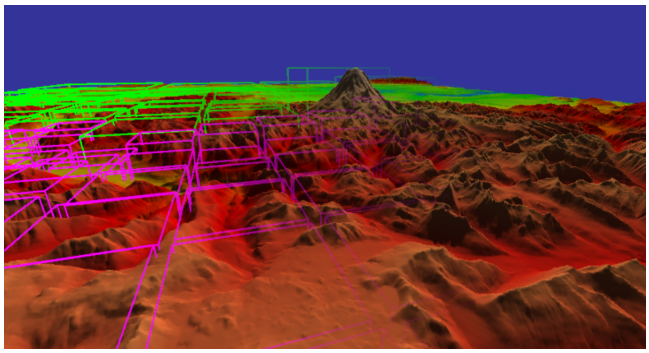
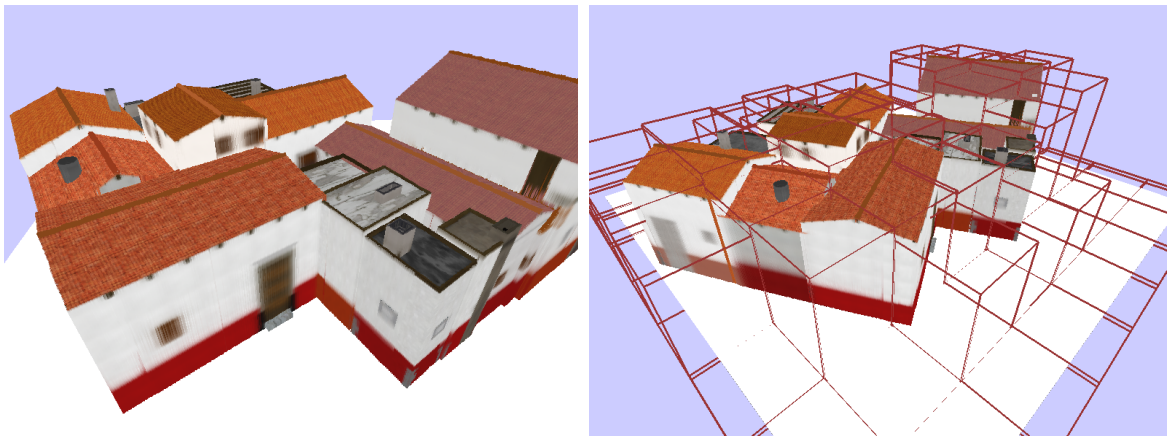


**Figure 7:** *Snapshot of an adaptive multiresolution rendering of a* $4k \times 4K$ *terrain (Pudget Sound model).*

**Figure 8:** *Four frames captured when visualizing High Quality Polynomial Texture Maps. Light position is bound to the position of the mouse on the window.*



(a) (b)

**Figure 6:** *Ray-Casted BlockMaps: (a) A fragment shader executes ray-casting over the BlockMaps of a city block at interactive rates (120 FPS). (b) a view of the hierarchical structures showing the bounding boxes of the hierarchy. The BlockMaps framework exploits the frustum culling and asynchronous data transfer facilities offered by the SpiderGL library.*

### Multiresolution for large data

Multiresolution comes into play whenever we want to show data that are too large to be handled w.r.t. the hardware capabilities. The building blocks of a multiresolution renderer are: a hierarchical layout of the data, an algorithm to visit such hierarchy and determine the nodes to use for producing the current frame, and the ability to load the nodes asynchronously, i.e. to proceed with rendering while missing data are being fetched.

Figure 7 shows a snapshot of a terrain rendering demo. The terrain is encoded by a quadtree of a $4K \times 4K$ texture storing elevation data. A regular grid of triangles is used to render each quad, by fetching the elevation of the vertices in a vertex shader.
Figure 6(a) shows a similar example where the potential of WebGL is even more evident. Here each quad contains a BlockMap [Di Benedetto et al. 2009], i.e. a small image encoding a portion of a urban dataset (both geometry and shading information) and the rendering is done by an ad-hoc ray casting algorithm implemented in the GPU as a fragment shader.
Figure 8 shows a multiresolution Polynomial Texture Map, which is an image where each pixel encodes the color as a function of light direction [Malzbender 2004]. Again, a quad tree is built from the the original $2930 \times 2224$ image and a fragment shader computes the current color as a function of the light position, passed as a global uniform variable. The size of the PTM is about 56MB since each pixels must store 9 values (3 for the RGB color and 6 for the PTM coefficients).

These examples show the potentiality of a WebGL-based application and the new possibilities opened by the availability of the GPU from within the web browser. SpiderGL eases the coding of applications like these by providing function for executing the frustum culling, computing on-screen projected error and handling asynchronous data transfer.

## 5 MeShade: deploying 3D content on the Web

While there are very large repositories for pictures, video or audio files, a web site like Flikr or YouTube for 3D models has yet to come. Up to now there are a few repositories of 3D models made by human modelers that one can browse and also few examples of repository of 3D scanned models [Stanford Computer Graphics Laboratory 2000; 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 [nex ] or by digital photography [Vergauwen and Gool 2006]) and for the ability to use 3D graphics hardware acceleration in the web browser.

MeShade is a web application 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 moveable panels (see Figure 9), 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. 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 uni-

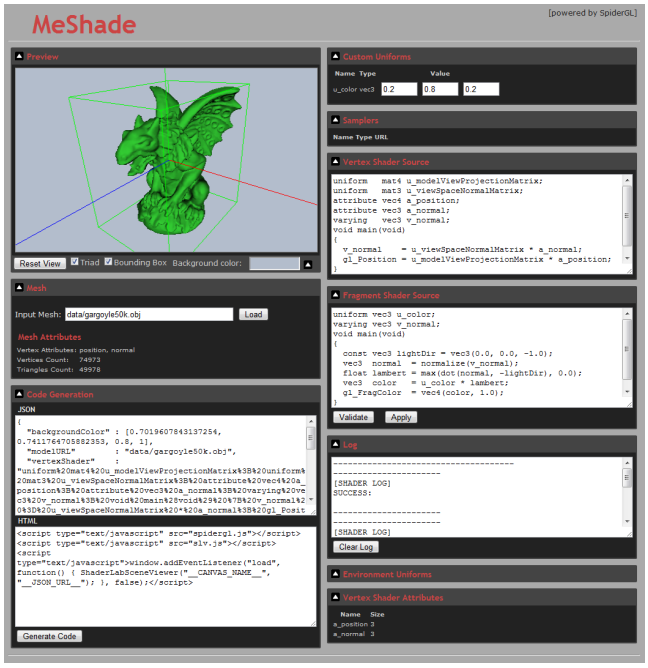



**Figure 9:** *The MeShade User Interface: once a 3D model URL is specified and data is loaded, the preview viewport displays the mesh using the vertex and fragment shaders written by the user on the page GUI itself. Edit forms for uniforms and texture images are dynamically added to the page as shaders are compiled.*

form; 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

In this paper we presented SpiderGL, a novel 3D Graphics JavaScript library which uses the upcoming WebGL specifications for realtime rendering. With practical examples, we have shown how the programming facilities exposed by the library help speeding up the creation of 3D web applications without forcing the developer to adopt predefined programming paradigms like scene graphs. By exploiting the JavaScript programming language to override the default behaviors of objects or express complex relations between the different entities involved in the rendering process, the developer is provided with high-level utilities for the creation of complex three-dimensional scenes, while retaining full access to the underlying WebGL layer. Common tools such as linear algebra, space related algorithms, asynchronous content loading and user interface utilities complete the main sectors the SpiderGL library addresses. The procedural style used throughout the library, coupled with mechanisms for object connections make the framework highly configurable and easily integrable into existing systems.
Furthermore we proposed a first potential application of SpiderGL that we called MeShade, to ease the deployment of 3D models on the web, which is likely to become a hot topic in a short while. All the material presented in this paper is freely available at `http://spidergl.org`

### 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.

### Acknowledgements

We would like to thank all the people at the Visual Computing Lab, ISTI - CNR, in particular Paolo Cignoni for his support and his advices.
The research leading to these results received funding from the EG 7FP V-City, The Virtual City (2008-11, IST-231199) and from the EG 7FP IP 3D-COFORM project (2008-2012, n. 231809).

## References

Microsoft Corp. `http://msdn.microsoft.com/en-us/library/aa751968(VS.85).aspx`.

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

APPLE CORPS. Squirrelfish : Webkit bytecode interpreter. `http://trac.webkit.org/wiki/SquirrelFish`.

BRUNT, P., 2010. GLGE: WebGL for the lazy. `http://www.glge.org/`.

CGAL PROJECT. CGAL, computational geometry algorithms library. `http://www.cgal.org`.

CROCKFORD, D. JSON (JavaScript Object Notation) . `http://www.json.org/`.

DELILLO, B., 2009. WebGLU: A utility library for working with WebGL . `http://webglu.sourceforge.org/`.

DELLEPIANE, M., CORSINI, M., CALLIERI, M., AND SCOPIGNO, R. 2006. High quality ptm acquisition: Reflection transformation imaging for large objects. In *VAST, 7th International Symposium on Virtual Reality, Archaeology and Cultural Heritage*, Eurographics Association, 179–186. Oct 30 - Nov 4, Cyprus.

DI BENEDETTO, M., CIGNONI, P., GANOVELLI, F., GOBBETTI, E., MARTON, F., AND SCOPIGNO, R. 2009. Interactive remote exploration of massive cityscapes. In *The 10th International Symposium on Virtual Reality, Archaeology and Cultural Heritage VAST (2009)*.

DON BRUTZMANN, L. D. 2007. *X3D: Extensible 3D Graphics for Web Authors*. Morgan Kaufmann.

FALCIDIENO, B. 2004. AIM@SHAPE project presentation. In *Shape Modeling International*, IEEE Computer Society, 329–338.

GOOGLE LABS . V8 JavaScript Engine . http://code.google.com/p/v8/.

GOOGLE LABS, 2009. O3D. http://code.google.com/apis/o3d/.

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

KAY, L., 2009. SceneJS http://www.scenejs. com

KHRONOS GROUP. GLU OpenGL Utility Library. http://www.opengl.org/documentation/specs/glu/glu1_3.pdf.

KHRONOS GROUP, 2009. Khronos: Open standards for media authoring and acceleration. http://http://www.khronos.org

KHRONOS GROUP, 2009. Webgl - opengl es 2.0 for the web. http://www.khronos.org/webgl/

KILGARD, M. J. GLUT - The OpenGL Utility Toolkit . http://www.opengl.org/resources/libraries/glut/

MALZBENDER, T. 2004. Enhancement of shape perception by surface reflectance transformation. In *Vision, Modeling, and Visualization*, 183.

MOZILLA, 2010. Tracemonkey: Mozilla javascript just-in-time compiler. https://wiki.mozilla.org/JavaScript:TraceMonkey

NextEngine. http://www.nextengine.com/.

RAGGETT, D. 1995. Extending WWW to support platform independent virtual reality. *Technical Report*.

RWTH. OPENMESH, visualization and computer graphics library. http://www.openmesh.org .

STANFORD COMPUTER GRAPHICS LABORATORY, 2000. http://graphics.stanford.edu/data/3Dscanrep/.

VCGLIB, visualization and computer graphics library. http://vcg.sourceforge.net.

VERGAUWEN, M., AND VAN GOOL, L. 2006. Web-based 3d reconstruction servic. *Machine Vision Applications 17*, 411–426.