Introduction to LiteGL

When developing OpenGL applications programmers have to remember lots of functions with complex names that receive several parameters of very specific types.
The nature of this syntax is to make it more powerful and flexible but could lead to very slow advance when developing common 3D applications.

This problem gets even worse in WebGL due to the nature of the Javascript language (soft-typing, garbage collection) and the browser restrictions (safety limitations, async calls).

The aim of LiteGL is to reduce this gap by wrapping most of the common WebGL 1.0 calls inside object oriented classes that represent clearer concepts (like Texture, Shader or Mesh).
And adding some very useful extra functions that most 3D application will eventually need (mesh parsing, texture copying, a pool of useful shaders,...).

Also it adds the necessary functions for any browser realtime application (canvas creation, input handling, events system).

Keep in mind that LiteGL wont free you from knowing WebGL, you still will need to do regular WebGL calls to handle the GPU attributes, or to do more specific actions on Textures, Meshes, etc.
But LiteGL should make it much easier to cope with the regular actions.

LiteGL has been using in several projects over the last 4 years with very good results. From weekend GameJam projects to professional applications or open source projects.

It is in a very mature state and almost 100% bug free.

Although I keep polishing it, the library is finished and no bigger changes are expected in the future (while we wait to WebGL 2.0 to be deployed globally).

LiteGL is based in LightGL.js by Evan Wallace, but some major changes were made to achieve better performance and clarity.

Dependencies

LiteGL only has one dependency, gl-matrix, which helps with all the mathematical operations common in all 3D applications. gl-matrix provides classes for vector3, matrix33, matrix44 and quaternions. And because it forces to use typed-arrays the performance is very good.

To better understand the syntax check the guide for gl-matrix.

Classes

There are four classes that any WebGL developer need to make any basic 3D application: Buffer, Mesh, Shader, Texture.

GL.Mesh and GL.Buffer

The GL.Mesh contains the geometry that must be rendered for an object.

It is just a container for several GL.Buffer which is the class that sends the data to the GPU, but GL.Mesh makes it easier to work with (loading, uploading to VRAM, generating new ones, etc).

There are also some methods to generate geometrical shapes like Spheres, Hemispheres, Boxes, etc. Check the examples to see all the shapes.

For more info read the guide about GL.Mesh and GL.Buffer

GL.Texture

The GL.Texture wraps a WebGLTexture. Helps to upload an image to the VRAM, apply FX or showing it to the viewport.

Because texture could come from several sources (images, video, canvas, data from memory) or in different forms (TEXTURE_2D, TEXTURE_CUBE_MAP) this class makes working with texture much easier.

For more info read the guide about GL.Texture

GL.Shader

GL.Shader wraps a WebGLProgram to it is easier to compile, check errors, bind or pass uniforms to the shader.

It also provides some basic shaders for copying data between textures.

For more info read the guide about GL.Shader

GL.FBO

Another useful trick in graphics imply rendering the scene inside a texture so it can be used in later stages, to do so you need a WebGLFramebuffer. The GL.FBO class wraps it so it is much easier to attach textures and enable it.

For more info read the guide about GL.FBO

Input and Mainloop

When creating an interactive application we need to ensure our code renders a frame constantly and handles the user input.

This is common in every WebGL application so LiteGL provides a convenient system to handle keyboard, mouse and gamepad inputs, and helps creating the main loop.

For more info read the mainloop and input guide

Helper classes

Besides the basic classes LiteGL comes with others that could help with more complex 3D applications.

GL.Octree and GL.Raytracer

This class helps testing ray collision against mesh in an efficient way. An octree is constructed containing all the mesh data so it can be crawled faster when testing collisions.

For more info read the Octree guide

geo

Following the gl-matrix coding style we provide a class to do basic collision detection between basic shapes (ray-sphere, ray-box, ray-plane, box-box)

For more info read the geo guide

LEvent

Additionaly LiteGL provides a simple but fast events system so you can dispatch events from any class and trigger callbacks.

For more info read the LEvent guide

Next steps

Once you know how to use LiteGL if you want to do more complex application you probably will need to use a scene graph with cameras and resources managers. If that is your case, I recommend you to check Rendeer.js (lightweight but fast renderer) and LiteScene.js (complex but powerful renderer), libraries build on top of LiteGL.

gl-matrix

When creating any 3D application the developer always need to have a vector3 class and a matrix4x4, mostly for screen projection and vertex transformations.

There are several mathmatical libraries for Javascript but all rely in creating classes to store every basic data, a class for vector3, a class for matrix44 and so on.

This has several problems, first the performance, and second the memory fragmentation.

To avoid this gl-matrix propose a different paradigm, instead of instantiating a class to store some data (like a vector3) we create the most basic Javascript typed array, a Float32Array with three elements.

This way accessing the data is fast and we can pack lots of data in one single array.

The only problem is that the syntax changes a little bit.

To create a var we have some functions:

```
//creating data in different ways
var vertex = vec3.create(); //it will contain [0,0,0]
var normal = vec3.fromValues(0,1,0); //it will contain [0,1,0]
var result = vec3.clone([1,2,3]); //it will contain [1,2,3]
//all three vars contain a Float32Array[3]
```

Instead of calling to a method in the var that contains the data, we call a global function and pass all the data it needs to perform the operation.

In gl-matrix the first parameter is always the place where to store the result of the operation.

```
//operating
vec3.add( result, vertex, normal ); //this adds vertex to normal and stores the result in result
```

When working with matrices is the same:

```
var matrix = mat4.create(); //the default is the identity matrix
vec3.transformMat4( result, vertex, matrix );
```

Keep in mind that LiteGL adds some extra useful functions to vec3, mat4 and quat.

For more info about gl-matrix check the documentation

GL.Mesh and GL.Buffer

To be able to render something we need to send the geometry of the object (vertices mostly, but could also be normals, texture coordinates,...) to the GPU.

Usually we store this information in buffers and send them so they are kept in the VRAM, this way the geometry can be easily retrieved by the GPU when it has to render the object.

To pack and send every buffer to the GPU you can use the class `GL.Buffer`.

Because normally an object is composed by several buffers, LiteGL uses the class `GL.Mesh` to store and manage `GL.Buffers`.

GL.Buffer

The class `GL.Buffer` contains several properties:

- **buffer**: this variable contains the `WebGLBuffer` handler of this buffer.
- **data**: this contains the typed array which contains a copy of the data that must be send to the GPU.
- **target**: the enum of WebGL to specify if this buffer is a GL.ARRAY_BUFFER (regular data) or a GL.ELEMENT_ARRAY_BUFFER (for indexing)
- **spacing**: tells every element in this array how many components does it have (a 3D point would have 3: (x,y,z)).
- **stream_type**: to tell the GPU how frequently the data will be modified, valid values are GL.STATIC_DRAW, GL.DYNAMIC_DRAW or GL.STREAM_DRAW
- **gl**: the WebGL context where this buffer is attached.

Some useful methods are:

- `upload( stream_type )`: this will take the data and send it to the GPU.
- `uploadRange( start, size )`: this will upload to the GPU only a range of the buffer (in case only some parts where modified)
- `applyTransform( mat4 )`: applies a matrix transformation to every vertex in this buffer. Used to bake transformations.

An example of how to create a buffer:

```
  var vertices_buffer = new GL.Buffer( gl.ARRAY_BUFFER, [0,0,0, 1,1,1, 0,1,0], 3 );
```

GL.Mesh

This is the container to store several buffers. You dont need to use this container but it helps loading and doing the render calls.

The important methods are:

- **vertexBuffers**: object that contains all the `GL.Buffer` by their name.
- **indexBuffers**: object that contains all the `GL.Buffer` that are for indexing.
- **info**: object that contain some extra info about this mesh (like groups).
- **bounding**: the bounding box in `BBox` format (center,halfsize,min,max,radius).
- **gl**: the WebGL context where this buffers were attached.

Some useful methods are:

- `upload`: uploads all the buffer again in the VRAM.
- `updateBounding`: recomputes the boundingbox from the vertices in the mesh.
- `getVertexBuffer( name )` returns the `GL.Buffer` with that name (not index buffers).
- `getIndexBuffer( name )` returns the `GL.Buffer` for index buffers.

How to create a mesh

You can create a mesh by creating every buffer individually and attaching them to a mesh or just using this method:

```
var mesh = GL.Mesh.load({
  vertices: [0,0,0, 1,0,0, 0,1,0],
  normals: [0,0,1, 0,0,1, 0,0,1],
  coords: [0,0, 1,0, 1,0],
  triangles: [0,1,2]
});
```

Basic primitives

If you want to generate procedurally some basic geometric shapes (like spheres, planes, cubes, etc) the class `GL.Mesh` comes with some handy functions.

You must call the function directly from the base class `GL.Mesh`(do not create the mesh using the ``new``` operator).

- `GL.Mesh.plane({...})`: to create a plane, params are { detail, detailX, detailY, size, width, heigth, xz (horizontal plane) }
- `GL.Mesh.cube({...})`: to create a cube, params are { size }
- `GL.Mesh.sphere({...})`: to create a sphere, params are { radius, lat, long, subdivisions, hemi }

An example:

```
var mymesh = GL.Mesh.sphere({radius: 10, subdivisions: 20});
```

Loading and Parsing

In case you want to load a remote mesh and parse it, the `GL.Mesh` comes with a basic OBJ loader but it can be extended to support other formats.

```
var mymesh = GL.Mesh.fromURL("meshes/mymesh.obj");
```

To extend the loader add your parser function to the `GL.Mesh.parsers[ format ]` container.

Textures

Using textures in WebGL is easy because the browser already comes with Image loaders.

The problems come when you want to apply very basic actions to the textures, like cloning, applying a shader to every pixel, resizing, etc.

For those reasons LiteGL adds many useful functions to the class `GL.Texture`.

Creating a texture

To create a texture just call the constructor:

```
var mytexture = new GL.Texture( 256, 256, { minFilter: gl.NEAREST, magFilter: gl.LINEAR });
```

As you can see the first parameter is the width, the second parameter the height, and the thirth parameter an object containing all possible options.

If no options are specified they will take the default value, which is:

- **texture_type**: gl.TEXTURE_2D, gl.TEXTURE_CUBE_MAP, default gl.TEXTURE_2D
- **format**: gl.RGB, gl.RGBA, gl.DEPTH_COMPONENT, default gl.RGBA
- **type**: gl.UNSIGNED_BYTE, gl.UNSIGNED_SHORT, gl.HALF_FLOAT_OES, gl.FLOAT, default gl.UNSIGNED_BYTE
- **filter**: filtering for mag and min: gl.NEAREST or gl.LINEAR, default gl.NEAREST
- **magFilter**: magnifying filter: gl.NEAREST, gl.LINEAR, default gl.NEAREST
- **minFilter**: minifying filter: gl.NEAREST, gl.LINEAR, gl.LINEAR_MIPMAP_LINEAR, default gl.NEAREST
- **wrap**: texture wrapping: gl.CLAMP_TO_EDGE, gl.REPEAT, gl.MIRROR, default gl.CLAMP_TO_EDGE (also accepts wrapT and wrapS for separate settings)
- **pixel_data**: ArrayBufferView with the pixel data to upload to the texture, otherwise the texture will be black
- **premultiply_alpha**: multiply the color by the alpha value when uploading, default FALSE
- **no_flip**: do not flip in Y when data is uploaded, default TRUE
- **anisotropic**: number of anisotropic fetches, default 0

Once the texture has been created if you want to change any of the properties you must do it manually calling the WebGL funtions.

Loading a texture from file

If you want to use an existing image as a texture you can use the loading methods for `GL.Texture`.

```
  var texture = GL.Texture.fromURL( "myimage.png", {} );
```

Cubemaps

Using cubemaps is very useful to create interesting effects but loading or generating is hard and prone to errors.

LiteGL comes with some useful functions to handle cubemaps, not only for loading but also to generate them from your scene.

```
var cubemap_texture = GL.Texture.cubemapFromURL( url ); //this assumes the url contains an image wit
```

Or if you have an image with a cross cubemap (aligned to the left)

```
var cubemap_texture = GL.Texture.cubemapFromURL( url, { is_cross: 1 } ); //for a cross image to the
```

Useful actions

Here is a list of useful methods that you can use with textures:

- `bind`: to bind it in one slot
- `clone`: returns a texture that its a clone of this one.
- `fill`: fills the texture with one solid color.
- `toViewport`: renders a full screen quad with the texture, check the documentation for more options.
- `applyBlur`: blurs the content of a texture, check the documentation to use it properly.

Render to texture

It is easy to render your scene to a texture instead of to the screen.
This is useful to create postprocessing effects or to bake information.

To render to one texture (or several) check the guide for the GL.FBO class.

Texture pool

When using temporary textures is always better to reuse old ones instead of creating and destroying them which would lead to garbage and fragmented memory.

LiteGL comes with a very simple texture pool system, you can release any texture or retrieve it.

Keep in mind that the textures pool would never be freed, so use it smartly.

```
//retrieve a texture from the pool
var temp_texture = GL.Texture.getTemporary( 256, 256, { format: gl.RGB } );
//here we use it...
//then release it so others can use it
GL.Texture.releaseTemporary( temp_texture );
```

Shaders

Shaders are fragments of code uploaded to the GPU that will be executed during the rendering process.
This way the programmer can control vertex projections and deformations or how every single pixel is illuminated.

Learning to code shaders is a complex subject that require good knowledge of mathmatics (algebra and 3d projection) plus a good understanding of how GPUs work.
This guide is not meant to teach you how to code shaders but to help you understand how to use your shaders in with LiteGL.

Please, if you never have coded shaders check any tutorial related of how to code shaders before going forward with this guide.

GLSL

In WebGL (as in OpenGL) all shaders must be coded in GLSL (GL Shading Language) which has a syntax similar to C++.

Every shader in WebGL 1.0 is made by two blocks of code, the Vertex Shader (executed per vertex) and the Fragment Shader (executed per pixel).

```
//Basic Vertex Shader
            precision highp float;
            attribute vec3 a_vertex;
            uniform mat4 u_mvp;
            void main() {
                gl_Position = u_mvp * vec4(a_vertex,1.0);
            }

//Basic Fragment Shader
            precision highp float;
            uniform vec4 u_color;
            void main() {
              gl_FragColor = u_color
            }
```

Note that the shader specifies the names of the streams (attributes).
In LiteGL the class GL.Mesh gives the streams some default names depending on the name of the buffer:

- **a_vertex** for vertices (vec3)
- **a_normal** for normals (vec3)
- **a_coord** for texture coordinates (vec2)
- **a_color** for vertex colors (vec4)
- **a_extra** for any extra stream of one single value (float)
- **a_extra2** for any extra stream of two values per vertice (vec2)
- **a_extra3** for any extra stream of two values per vertice (vec3)
- **a_extra4** for any extra stream of two values per vertice (vec4)

Create a shader

To create a shader un LiteGL you need to use the class `GL.Shader`.

Here is the different ways you can create a shader:

If the code is stored in two variables:

```
var myshader = new GL.Shader( vertex_code, fragment_code );
```

Sometimes you want to compile the same shader but with different pre-processor macros, then you can pack the macros inside an object and pass it as thirth parameter:

```
var macros = { USE_PHONG: "" }; //this will be expanded as #define USE_PHONG
var myshader = new GL.Shader( vertex_code, fragment_code, macros );
```

Or if the code is in two files:

```
var myshader = GL.Shader.fromURL( vertex_code_url, fragment_code_url );
```

Or you can use the `GL.loadFileAtlas(url, callback)` which allow to load one single file that contains the code of all the shaders easily.

Setting up the uniforms

To use a shader you need to upload some variables from javascript to be used in the computations:

```
//in bulk
myshader.uniforms({ u_mvp: matrix, u_color: [1,1,1,1] });

//per uniform
myshader.setUniform("u_color",[1,1,1,1]);
```

Information about the shader

You can extract information about the compiled shader like which uniforms or attributes does it use and the types:

```
//if shader contains this uniform
if( myshader.uniformInfo["u_color"] )
    //...
```

Render using this shader

If you have already your GL.Mesh you can render it using a shader with the next command:

```
   myshader.uniforms({u_mvp:matrix}).draw( mymesh );
```

Default Shaders

The system allows to create some basic shaders easily:

```
var flat_shader = GL.Shader.getFlatShader(); //useful for rendering flat lines
```