Guide to develop for LiteScene

This guide intends to help people understand the engine so they can take full advantage of it from the WebGLStudio platform.

The most important thing to understand is that the engine is separated in several layers, every one of them is independent, so to better understand everything about LiteScene please first go to LiteGL.js which is the low-level layer in charge of accesing WebGL, and the one LiteScene uses to simplify the GPU calls.

Guides

Here there is a list with the most commont topics to master LiteScene:

- Scene: Understanding the scene tree
- Components: How to use the components system
- Scripting: How to create your own scripts
- Input: how to get user input
- GUI: how to add a GUI to your application
- Resources: How to handle resources (textures, meshes, etc)
- Player: how to use the player to embed your scene in any website
- Custom components: How to create your own components

Some advanced topics:

- Operating with Vectors and Matrices: understanding gl-matrix and how to do mathematical operations.
- Events: how to capture events from the system to call your own callbacks
- Render pipeline: How does the render pipeline work
- Shaders: How to write your own shaders
- Post-processing: How to apply postprocessing effects
- Animation: How to create animations
- Custom editor interfaces: configuring the interface in the editor
- Tweening: how to interpolate values easily
- Draw: how to draw in immediate mode

Index

Features

LiteScene is an engine meant to work with WebGLStudio although it is not mandatory to do so (you can create full scenes just by accesing the engine directly from code).
The engine is meant to be very modular and easy to extend.

Simple things can be added to the engine (like new modules or materials) without really needing to understand the whole system, but for more complex behaviours you will need to learn the system as a whole.

Because the whole engine is coded in Javascript (without any transpiler) you have full access from within the engine to any part of it, that means that you could create a script that replaces the behaviour of an existint part of the engine without any problem, thanks to the nature of Javascript and the modularity of the system.

The engine also allows different ways to do the same actions to help people, so you can use in-editor scripts, or external scripts loaded on launch, or graphs, or directly replace the source code of the engine.

The engine is also meant to help the editor to understand what is going on in every component, this way users can create new components with personalized interfaces that helps them to setup their scenes, without the need to code them.

Limitations

LiteScene is not meant to be used as a powerful 3D engine, it has its limitations regarding to number of objects in the scene or complexity of the interactions. If should be used in simple scenes with tens of objects at most.

LS Namespace

The LS namespace is the global namespace where all the classes are contained, some generic methods are also here that could be helpful to control the global system.

Check the documentation for a complete list with all the methods.

Inside LS there are some important object that you should be aware:

- Components
- MaterialClasses
- Formats

Some of the most important components (such as Script, Camera, Light and Transform) are stored also in the LS namespace besides being in LS.Components.

SceneTree and SceneNode

To handle the objects in a scene the user must understand how to use the SceneTree and the SceneNode object.

While SceneNode represent an object in the scene, SceneTree represents the scene itself.

Every node could contain other nodes as children similar to how the DOM works.

The SceneTree contains a root node (scene.root) where all the nodes in the scene are pending.

For more info about the Scene read the Scene guide.

Components

The behaviour of every node comes from the components attached to it.

Cameras, Lights, MeshRenderers, etc, are all components that could be attached to any SceneNode to add functionalities.

For more info about the Scene read the Components guide.

Scripts

Scripts are a special component that contain code that should be executed. This is done to allow scenes to contain behaviour written by its author without the need to include external libraries.

For more info about the scripts read the Scripting guide.

Graphs

Graphs are another interesting component similar to scripts because they allow the creator to insert some behaviours, but instead of using code it uses graphs made combining nodes.

For more info about the scripts read the Graphs guide.

Renderer

One of the main purposes of LiteScene is to provide a reliable render pipeline that can handle most common scenarios (multi-texture materials with several light sources, shadowmaps and reflections) but also giving total freedom to create your own rendering pipeline if you want.

For this reason there are several classes that handle the rendering process like `LS.Renderer`, `LS.RenderInstance`, `LS.RenderState`, `LS.ShaderCode`, `LS.RenderFrameContext` plus the obvious ones: `LS.Material`, `LS.Light`, `LS.Camera` that could be tweaked in many ways.

For more info about the rendering pipeline read the Rendering guide.

ResourcesManager

There are many resources used by the application like Textures, Meshes and Shaders for rendering the frame, but also Scripts for behaviour, audios or data files.

All this resources are loaded and stored in the ResourcesManager.

For more info about the rendering pipeline read the Resources Manager guide.

Player

Once a scene has been created using an editor like WebGLStudio you want to embed it easily in your website without having to handle all the events and the main loop manually. For this reason there is the `LS.Player` class, that handles all the low-level actions plus loading and starting the scene.

For more info about the player read the Player guide.

Network

To handle requesting files from the network we supply with a high-level class called Network.

Actions

To make the editor more powerful some components allow to perform special actions on them directly from the editor (they appear in the contextual menu of the component if you right click in the title).

This actions could be defined by the components, or by the editor, and stored in a way they can be easily accessed.

Here are some examples of common actions:

- copy
- paste
- reset
- share

But some components could have special ones.

Action object

An action is an object with some properties:

- title: what text to show to the user when listing actions
- callback: callback to call when the action is performed (the this will the component where the action is performed)
- callback_show: [optional] a callback that must return true if this action must be shown to this component

How to define actions

There are three ways to define actions:

- In the class Component: actions that are global to any component
- In the constructor of a specific component: this action is specific of this component class
- In the instance of a component: this action is specific of this instance

To add them to the component class:

```
LS.Component.actions["copy"] = {
    title:"Copy",
    callback: function(){
        EditorModule.copyComponentToClipboard(this);
    }
};
```

To add them to an specific component:

```js
LS.Components.Camera.actions["setview"] = {
title: "Set to view",
callback: function() {
//...
}
};
```

Or to add to an instance:

```
myinstance.getActions = function( actions )
{
//modify actions array
//...
return actions;
}
```

Animation

LiteScene allows to animate any property of any component or node in the scene.

This is used for the skeletal animation but it can be used to animate other properties like camera position, material properties or any field that the user wants.

The animation system works by storing tracks with keyframes that contain the time and the values to apply.

When we want to apply an animation we use the PlayAnimation component.

You can use the WebGLStudio timeline editor to edit any animation in the scene.

The LS.Animation and LS.Animation.Take

Animations are stored in a big container called LS.Animation that behaves like a resource.

Instead of storing the tracks per animation, we store them in another container called Take, this way one animation could contain several subanimations (takes).
By we usually use the default take.

Every Take contains a list of tracks, and the total duration of the take.

Because every scene usually needs to have an animation track, to make it easier, you can have one global animation track stored in the scene itself.

To create it you can call `LS.GlobalScene.createAnimation()` and this track will be saved with the scene.

LS.Animation.Track

Every track contains all the info to modify one property of the scene as time goes.

They contain a locator, a list of keyframes, and information about the interpolation method.

There are two types of track:

- Property tracks: every keyframe represents a value to assign to the property specified in the track locator.
- Event tracks: every keyframe contain info about an event or a function call that should be performed

Locators

Every track has a string called the locator which identifies the property in the scene affected by the animation track.

Some examples: `root/transform/x`,`mynode/MeshRenderer/enabled` or `@NODE-f7cac-865-1ecf644-5\@COMP-f7cac-865-1ecf644-3\size`.

The locator is usually (but not always) divided in three parts:

- node: could be the UID or the name of the node
- componentt: to specify which component, could be the UID or the name of the component
- property: to specify the name of the property

Some components handle the locators by themselves (like script components) because they allow more parts in the locator.

To get the locator of a property you can call the method getLocator of the container (the component) passing the name of the property as a parameter:

```
node.transform.getLocator("x"); //returns "@NODE_uid/@COMP-uid/x"
```

PlayAnimation

To play an animation you must use the PlayAnimation component.

This component allows to select the animation, choose the take, the playback speed, and different modes.

If you want to use the global scene animation leave the animation field blank (or use "@scene").

Components

Every node could host several components.

A component is a element that adds behaviour and visual representation to a node. There are lots of different components that can be attached to any node to add behaviour.

Thanks to components the engine is very dynamic, allows to construct new behaviours by blending different components in one node, and allows to extend the system with ease.

All the component classes are stored in `LS.Components`.

Instantiating and attaching components

To create a component you just instatiate the class:

```
var my_component = new LS.Components.Camera();
node.addComponent( my_component );
```

To access the component:

```
var my_component = node.getComponent( LS.Components.Camera );
```

or to remove it

```
node.removeComponent( my_component );
```

Check the documentation for more info.

Creating your own components class

If you want to create new component classes check the guide to programe new components.

Important Components

There are several components that are very important for any scene, they are:

- **Transform**: to handle the position, rotation and scaling of every object in the scene.
- **Camera**: to choose from where to render the scene and how.
- **MeshRenderer**: to render something in the scene.
- **Light**: to iluminate the scene.

Transform

Transform is the component in charge of handling where a SceneNode is located spacially on the scene, where is heading and which is its size.

It also provides methods to handle the different coordinates systems.

Camera

The Camera component adds a camera to the scene, which will be used by the Renderer to create a representation on the screen.

MeshRenderer

MeshRenderer is the component in charge of rendering meshes into the scene.

Light

Light is the component in charge of holds the info about the light sources. The behaviour depends on the rendering engine and shaders.

Script and ScriptFromFile

To add extra behaviour there are components that could be programmed from the editor.

For more info read the Scripting guide.

Editor Interfaces

When creating components you usually want to control their data from WebGLStudio editor,
this means that you need to know a little bit about the architecture of WebGLStudio in order to create user interfaces.

LiteGUI

All the interfaces in WebGLStudio are created using the library LiteGUI, so it is important to know a little bit how it works in order to take full advantage.

But the most important element is to know the class `LiteGUI.Inspector` which is in charge of create the widgets.

To know more about the Inspector, check the guide about LiteGUI.Inspector

Public Vars

WebGLStudio creates a default interface for every public var contained inside a Component unless an specific function is created.

The problem with automatically generated interfaces is that they cannot know the true meaning of the vars to adapt the widgets. An String could be a text, a url, a resource path, a script, etc.

This is why LiteScene allows to define properties per variable:

```
function MyComponent(o)
{
  this.myvar = null;
}

MyComponent["@myvar"] = { type: "texture" };
```

Or even specify properties:

```
MyComponent["@myvar"] = { type: "Number", widget: "slider", min: 0, max: 100, step: 1 };
```

Creating custom interfaces

But if you dont want to use the automatic interface, you can redefine by creating the inspect function:

```
MyComponent["@inspector"] = function( component, inspector )
{
  inspector.addString("Name",this.myvar, { callback: function(v){ component.myvar = v; } });
}
```

Or if you want to add something extra to the default interface generated by the system:

```
MyComponent.onShowProperties = function( component, inspector )
{
  inspector.addInfo( null, "This component is interesting" );
}
```

LS.Draw

To render things on our scene we rely on the rendering pipeline and the LS.RenderInstances that are collected from the scene.

But sometimes we want to render in immediate mode (mostly for debug purposes but also to create 3D GUIs or helpers).

LiteScene includes a namespace called `LS.Draw` which contains several functions to render basic shapes or meshes without the need to create a LS.RenderInstance.

Keep in mind that anything rendered using immediate mode wont receive any of the behaviours provided by the render pipeline (like shadows, reflections, picking, etc).

Also remember that you can always call WebGL directly or to use the LiteGL methods to access WebGL to render anything, LS.Draw is just a list of helpers to make it easier.

Rendering in immediate mode

To render in immediate mode first you need to be sure that your code will be executed during the rendering loop.

To ensure that, you must place your code inside a function call during the rendering loop like the `onRender` on a script:

```
this.onRender = function()
{
   LS.Draw.setColor([1,1,1]);
   LS.Draw.renderSolidBox( 10,10,10 );
}
```

Or if you are using your own component you can bind any of the rendering methods like `beforeRenderInstances` or `afterRenderInstances`.

Basic shapes

When you have some basic data like points or lines use:

- `renderPoints`: renders the points on the screen using the point size set by `LS.Draw.setPointSize`, you can pass also a stream with colors if they have different colros.
- `renderPointsWithSize`: same as `renderPoints` but you can pass an extra stream with the size of every point related to the global point size.
- `renderRoundPoints`: same as `renderPoints` but points will be rounded.
- `renderLines`: renders lines passed as an array of points, you can specify LINE_STRIP if you want.

```
LS.Draw.renderPoints( [0,0,0, 100,100,100], [1,1,1,1, 1,0,0,1] ); //this will render two points, one
LS.Draw.renderLines( [0,0,0, 100,100,100] ); //this will render one line
```

If you have a mesh you want to render just call the `renderMesh` function:

```
LS.Draw.renderMesh( mymesh, GL.TRIANGLES ); //check the documentation if you want to render a range
```

But there are also some basic shapes that you can render without the need to create the Mesh, like basic shapes:

- `renderCircle ( radius, segments, in_z, filled )`
- `renderCone ( radius, height, segments, in_z )`
- `renderCylinder ( radius, height, segments, in_z )`
- `renderRectangle ( width, height, in_z )`
- `renderSolidBox ( sizex, sizey, sizez )`
- `renderSolidCircle ( radius, segments, in_z )`
- `renderWireBox ( sizex, sizey, sizez )`
- `renderWireSphere ( radius, segments )`

Other useful functions are:

- `renderText`
- `renderImage`

Applying transformations

All functions will render the object in the current center of your coordinates system.
To change the coordinates system you can apply basic transformations:

```
LS.Draw.translate(10,10,10);
LS.Draw.rotate(90,0,1,0);
LS.Draw.scale(2,2,2);
```

And if you want to save and retrieve the state of the transformations you can use `push` and `pop`:

```
LS.Draw.push(); //saves coordinates system
LS.Draw.translate(10,10,10);
LS.Draw.renderCircle( 10,10 );
LS.Draw.pop(); //recover old coordinates system
```

Global properties

Every render will try to use the global properties to define the color, alpha and point size:

```
LS.Draw.setColor([1,0,0]); //you can pass a fourth parameter with the alpha
LS.Draw.setAlpha(0.5);
LS.Draw.setPointSize(10); //in pixels
```

Camera

Also if you want to change the camera position you can also pass a camera although the system already does it:

```
LS.Draw.setCamera( mycamera );
```

Documentation

Check the LS.Draw documentation for better explanation of every function.

Events

The LiteScene system was designed so it is fairly easy to add new features or modify the existing ones.

To accomplish this the system relies mostly in events dispatched at certains points of the execution.
This way is the user wants to intercept the execution flow and add some steps it can be done.

Events are triggered from different elements of the system:

- from the scene (`LS.GlobalScene`): when they are related to the scene (like update, start, etc)
- from the renderer (`LS.Renderer`): when they are related to the rendering
- from the resources manager (`LS.ResourcesManager`): when they are related to the loading of resources
- from the node itself (`LS.SceneNode`): when is events related to this node

This events are triggered in an specific order, it is important to understand that to be able to usem them for your needs.

Binding Events

In case you want to bind some actions to an event you must bind the callback to that event:

```
LEvent.bind( LS.GlobalScene, "update", my_update_callback, my_instance );
```

And to unbind:

```
LEvent.unbind( LS.GlobalScene, "update", my_update_callback, my_instance );
```

As you can see we are passing not only the callback but also the instance as a parameter.
This is important because this way we can unbind it without problem.

Warning: Using tricks like the `Function.prototype.bind()` method would create a new callback every time so we wouldnt be able to unbind it from the system again. Avoid to use it when passing callbacks to events.

Execution Events

Here is a list of events and in which order they are called.
They are all triggered from the current scene (`LS.GlobalScene`):

- "start": when the application starts after loading everything
- *Main Loop*
  - *Rendering process*
  - "beforeRender"
  - *data is collected from the scene*
  - "afterCollectData": collect all the render data (cameras, render instances, lights)
  - "prepareMaterials": in case you want to edit material properties
  - "renderShadows": to render shadowmaps
  - "afterVisibility": in case we want to cull object according to the main camera
  - "renderReflections": to render realtime reflections to textures
  - "beforeRenderMainPass": after all things set, before we render the scene
  - "enableFrameContext" to enable the render output context
    - *For every camera*
    - "beforeRenderFrame"
    - "enableFrameContext" (on the camera) to enable the render output context
    - *Clear buffer*
    - "beforeRenderScene"
    - "beforeRenderInstances"
    - *the scene instances are rendered*
    - "renderInstances"
    - "renderScreenSpace"
    - "afterRenderInstances"
    - "afterRenderScene"
    - "afterRenderFrame"
  - "showFrameContext": show the frame in the screen
  - "renderGUI": render 2D content
  - "afterRender": after all the scene has been rendered in the screen
- "beforeUpdate" before the update
- "update" during the update
- "afterUpdate" after the update

- "finish" when the application stops (in the editor)
- "pause" if the editor pauses the execution
- "unpause" if the editor unpauses the execution

File Formats and Parsers

LiteScene supports several file formats to store Meshes, Textures and Data from the Scene.

But the idea of LiteScene is to make it easy to add support to new file formats just by adding the file format parser to the system.

There are some classes in the system in charge of loading and parsing, they are `LS.ResourcesManager` (to load, process, and store files), and `LS.Formats` (to store information about how to parse a file).

To understand better the file parsing we need to see the steps taken by the LS.ResourcesManager to load a file.

How the resources loading works

- We call `LS.ResourcesManager.load` passing the url of the resource we want to load.
- `load` will check the file extension info using `LS.Formats.getFileFormatInfo( extension );` to see if it has to force a dataType in the request. This will check for the info registered with this file format (the one passed to `LS.Formats.addSupportedFormat`).
- Once the file is received it will be passed to `LS.ResourcesManager.processResource` to be processed.
- If the resource extension has a preprocessor callback it is executed. A preprocessor is a function that takes the data requested and transforms it to make it ready to be used.
- If the preprocessor returns true it means it has to wait because the processing is async, once finished it will call `processFinalResource`
- If no preprocessor:
- If if is a Mesh calls `processTextMesh` which will call to the file format `parse` function
- If it is a Scene calls `processTextSCene` which will call to the file format `parse` function
- If it is a Texture calls `processImage` which will call to the file format `parse` function
- If the format_info has a parse method call it
- If it is neither of those types then it is stored as a plain `LS.Resource` asumming it is just data.
- Once processed it calls `processFinalResource` which is in charge of storing the resource in the adequate container.
- Then the resource is registered in the system using the `registerResource` function
- Which will call to its postprocessor callback if it has any (mostly to store the resource propertly, compute extra data, etc)

All this steps are necesary because different types of resources require different actions, and also because different file types share the same actions.

Guide to add new file formats

Adding new file formats requires several steps:

- Creating an object with all the info for the file format like:
- **extension**: a String with the filename extension that is associated with this format (or comma separated extensions)
- **type**: which type of resource ("image","scene",mesh"), otherwise is assumed "data"
- **resource**: the classname to instantiate for this resource (p.e. Mesh, Texture, ...)
- **dataType**: which dataType send with the request when requesting to server ("text","arraybuffer")
- **parse**: a callback in charge of converting the data in something suitable by the system. If null then the data will be as it is received.
- Registering it to LS.Formats with `LS.Formats.addSupportedFormat( "extension", MyFormatInfo );`
- If you want a preprocessor you need to call to `LS.ResourcesManager.registerResourcePreProcessor(extension, callback)`
- If you want a postprocessor you need to call to `LS.ResourcesManager.registerResourcePostProcessor( resource_classname, callback)` but that shouldn't be necessary because all resources have already its own postprocessor.

Example of adding support for a new fileformat

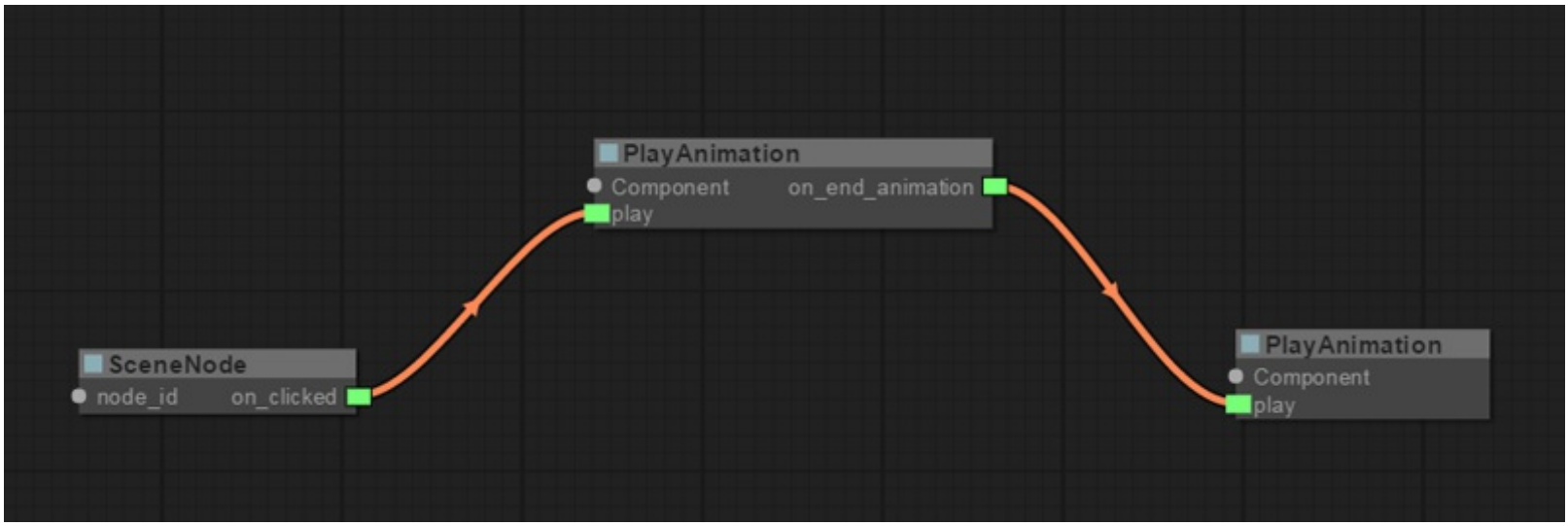
```
//this will allow to load SRTs as text objects instead of binary objects
LS.Formats.addSupportedFormat( "srt", {
   extension:"srt",
   dataType: "text"
});
```

Graphs

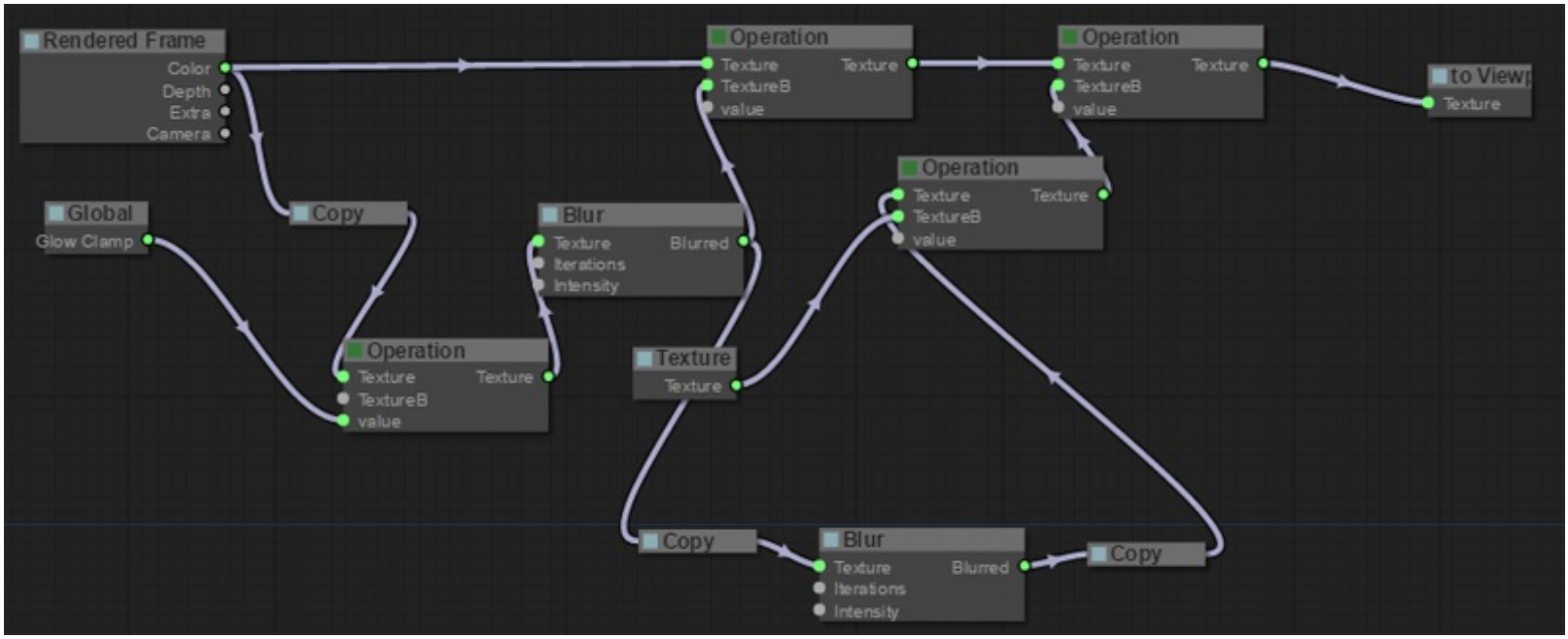
Besides using Scripts to control behaviour and rendering passes, you can also use Graphs.

A Graph contains nodes representing elements of your scene, and wires to transfer actions or data between the nodes.

This way you can set up simple behaviours without the need of using code and in a more transparent way.



You can also use graphs to control the rendering postprocessing effects using the FXGraphComponent.



To know more about graphs check [LiteGraph.js]

Creating a GUI on top of your app

In this chapter we will discuse different ways to create a user interface for you application.

HTML GUIs

Usually when people start using LiteScene they wonder if the library includes a GUI system to create the interface of their applications. But they ignore that the web already has the best GUI system possible, and it is called HTML.

The only problem is that it requires to know HTML and how the DOM works.

Use HTML GUIs if you have to display lots of information, or if you want to use some special character set (japanese characters, etc).

Check the guide to create GUIs in HTML for a tutorial on how to use it.

Immediate GUI

The immediate GUI is a set of functions made to help create GUIs very easily. It is similar of how Unity does it.

It works by calling some `LS.GUI` methods from the `onRenderGUI` (or the `renderGUI` event). Most methods require to pass the area `[x,y,width,height]` for the widget, and catching the returned value in case of any interaction expected.

The current widgets are:

- `LS.GUI.Box( area, color )`: to draw a box in the GUI
- `LS.GUI.Label( area, content )`: to draw a text label (or an image) on the GUI
- `LS.GUI.Button( area, content, content_over )`: to draw a button (content could be some string or a GL.Texture). Returns true if the button has been pressed.
- `LS.GUI.TextField( area, text, max_length, is_password )`: to draw a text field widget. Returns the text.
- `LS.GUI.Toggle( area, value, content, content_off )`: to draw a checkbox (content could be some string or a GL.Texture). Returns the current value of the checkbox (it will be the same as the one passed unless the user clicked the toggle).
- `LS.GUI.HorizontalSlider( area, value, left_value, right_value, show_value )`: to draw a slider (left_value is min, right_value is max). Returns the current value of the slider (it will be the same as the one passed unless the user clicked the slider).
- `LS.GUI.VerticalSlider( area, value, left_value, right_value, show_value )`: to draw a slider (left_value is min, right_value is max). Returns the current value of the slider (it will be the same as the one passed unless the user clicked the slider).

As you can see some widgets (like Toggle, TextField or the sliders) return the resulting value. It is important that the returned value gets passed again the next time the widget is rendered. Otherwise the changes wont affect the widget.

There are some interesting variables to tweak the GUI:

- `LS.GUI.GUIOffset` is a vec2 that can be changed to position the GUI somewhere else. BUt it must be set every frame.
- `LS.GUI.GUIStyle` contains some vars for colors and properties used to stylize the GUI.

Here is an example of immediate GUI with all the widgets from a Script:

```
//@immediate gui
//defined: component, node, scene, globals

this.createProperty("texture","", LS.TYPES.TEXTURE );

this.toggle_value = true;
this.slider_value = 50;
this.text = "text";

this.onRenderGUI = function(ctx)
{
    LS.GUI.Box( [5,5,320,600], "#111" );

    LS.GUI.Label( [10,10,300,50], "Example of GUI" );

    if( LS.GUI.Button( [10,80,300,50], "Pulsame" ) )
    {
        console.log("pulsado!");
    }

    this.text = LS.GUI.TextField( [10,140,300,50], this.text );
    if( LS.GUI.pressed_enter )
            this.text = "";

    this.toggle_value = LS.GUI.Toggle( [10,200,300,50], this.toggle_value, "toggle" );

    this.slider_value = LS.GUI.HorizontalSlider( [10,260,300,50], this.slider_value, 0,100, true );

    for(var i = 0; i < 10; ++i)
        this.slider_value = LS.GUI.VerticalSlider( [15 + i * 30,320,24,100], this.slider_value, 0,10(

    LS.GUI.Label( [20,440,100,100], LS.RM.textures[ this.texture ] );
}
```

Canvas GUIs

The last option is to create the GUI manually, you can use Canvas2DToWebGL, a library that allows to use regular Canvas2D calls inside the WebGL Canvas, this way you can easily render simple GUIs (more suited for non-interactive GUIs like HUDs).

Also one of the benefits of using Canvas2D is that you can render WebGL textures (even the ones generated by the engine) as part of the interace.

```
this.onRenderGUI = function()
{
  var ctx = gl;
  ctx.start2D();
  ctx.fillStyle = "red";
  ctx.font = "20px Arial";
  ctx.fillText( "Hello", 100,100 );
  ctx.finish2D();
}
```

And if you want to check if the mouse is in a screen position you can use some of the `LS.Input` functions:

```
this.onRenderGUI = function()
{
  var ctx = gl;
  ctx.start2D();
  if( LS.Input.Mouse.isInsideRect( 100, 100, 120,80, true ) )
  {
    ctx.canvas.style.cursor = "pointer";
      ctx.fillColor = [0.5,0.5,0.7,0.75];
  }
  else
  {
        ctx.fillColor = [0.2,0.2,0.3,0.75];
        ctx.canvas.style.cursor = "";
  }
  ctx.fillText( "Hello", 100,100 );
  ctx.finish2D();
}
```

Check `LS.Input` guides for more info of how to handle input.

GUI in HTML

When you want to have a complex GUI in your app the best option is to use HTML.

HTML is well documented, powerful, widely known, has lots of great features and it merges perfectly with the WebGL Canvas.

So when creating applications that require the user to interact with some widgets the fastest solution is to use regular HTML elements on top of the WebGL Canvas.

Maybe there is a slight performance drop due to the way browsers compose WebGL and HTML together, but it is the best way to ensure the GUI will fit well in a website.

HTML and LiteScene

You are free to add any HTML element to the DOM at any moment, there are no limitations. The only problem come when you use the editor WebGLStudio.

When working with a scene in WebGLStudio you have to ensure that your HTML is not colliding with any of the editor, and that your HTML will be destroyed once the scene is reset.

To do so you must follow some rules that are explained in the next chapters.

Creating HTML Elements

First, do not add HTML elements to the GUI till the application starts (when the onStart event is launched), otherwise there will be elements during the editing process that could interfere with the user.

Attaching HTML to the DOM

Second, all DOM elements must be attached to the GUI root element, this way the system can remove all DOM elements if the scene is reset.

To get the GUI root element use the function `LS.GUI.getRoot()`:

```
var gui_root = LS.GUI.getHTMLRoot();
gui_root.innerHTML = "<button>click me</button>";
```

Be careful changing the style of the GUI root element. The GUI root element has the classname "litescene-gui", use that class to encapsulate your CSS properties to avoid collisions with other elements in the DOM.

Mouse Events

The GUI root element is a div with 100% width and 100% height, positioned so that it overlays completly the canvas.

Because of that the div would get all the mouse events and no mouse events would reach the canvas.

To avoid that the GUI element has the `style.pointerEvents` set to `none` which means all the mouse events will be ignored by the element **and all its children**.

But sometimes we want to process mouse events in our widgets (clicking buttons, draging scrolls, etc).
If that is the case, you must set the `style.pointerEvents` to `auto` in the element that could get mouse events. You could do it by code or by CSS.

Keep that in mind when working with GUI elements.

Creating HTML panels

Most of the time you would like to make just a floating div on top of the GUI to show some HTML information.
To help in those cases the easier solution is using the `LS.GUI.createElement` that behaves similar to `document.createElement`, creating a tag of a given parameter, but it has some extra functionalities.

First the element will be attached automatically to the GUIElement so we dont have to do it.

Second the element pointerEvents will be set to auto, so we can add any normal HTML code and have the expected behaviour.

Thirth, it will be anchored to the corner of our canvas that we specify (if not anchor point is specified it will asume top-left).

Here is an example :

```
var panel = LS.GUI.createElement("div","top-left");
panel.innerHTML = "<h3>HELLO!</h3>";
```

Valid anchors are "top-left","top-right","bottom-left" and "bottom-right";

DOM events

From here if you want to bind events to the DOM elements you can do it as in any HTML website:

```
var button = panel.querySelector("button");
button.addEventListener("click", my_function );
```

Using HTML files

Creating HTML from the javascript code is very tedious and not very friendly.

Usually when creating HTML code you want to write it inside a normal HTML file. You can create a text file from the editor in WebGLStudio and add all the HTML code inside the file, retrieve the file when your application starts and attach it to the GUI element.

Here is one example:

```
this.onStart = function()
{
  LS.ResourcesManagear.load("myfile.html", function(res){
    var gui_root = LS.GUI.getHTMLRoot();
    gui_root.innerHTML = res.data;
  });
}
```

Or you can store several HTML elements and attach the ones that you need by retrieving it as a HTML element and using selector querys:

```
this.onStart = function()
{
  LS.ResourcesManagear.load("myfile.html", function(res){
    var gui_root = LS.GUI.getHTMLRoot();
    var html = res.getAsHTML();
    gui_root.appendChild( html.querySelector("#mypanel") );
  });
}
```

Or to go faster just attach them as you need:

```
this.onStart = function()
{
  LS.ResourcesManagear.load("myfile.html", function(res){
    var html = res.getAsHTML();
    LS.GUI.attach( html.querySelector("#mypanel"), "top-left" );
  });
}
```

Or if you want to attach all the HTML to the GUI you can use `LS.GUI.load("myfile.html", my_function)`:

```
this.onStart = function()
{
  LS.GUI.load("myfile.html", function(html_root){
    html_root.querySelector("#mypanel").addEventListener("click", my_click_function);
  });
}
```

Input

In this guide we will explain different ways to read the user input

Input Events

The Event driven input is meant to capture events that happend when the user interacts with the application, like keys pressed, mouse movements, buttons being pressed, etc.

If you are using a Script component you can catch the events easily using the auto-binded callbacks like:

- `onMouseDown`: when a mouse button is pressed
- `onMouseUp`: when a mouse button is released
- `onMouseMove`: when the mouse moves
- `onMouseWheel`: when the mouse wheel rotates
- `onKeyDown`: when a key is released
- `onKeyUp`: when a key is pressed
- `onGamepadConnected`: when a gamepad is connected
- `onGamepadDisconnected`: when a gamepad is disconnected
- `onButtonDown`: when a gamepad button is pressed
- `onButtonUp`: when a gamepad button is released

All this functions receive the system event by parameter.

```
this.onKeyDown = function(event)
{
    if(event.keyCode == 39)
        console.log("Right");
}

this.onMouseDown = function(event)
{
    if( event.which == GL.LEFT_MOUSE_BUTTON )
        console.log("Left mouse button pressed!");
}
```

Sometimes we are coding our own component for the Components pool. In that case we have to bind the events manually:

```
this.onAddedToScene = function(scene)
{
    LEvent.bind( scene, "mousedown", this.myMouseCallback, this );
}

this.onRemovedFromScene = function(scene)
{
    LEvent.unbind( scene, "mousedown", this.myMouseCallback, this );
}

this.myMouseCallback = function(type, event)
{
    console.log("mouse button clicked:", event.button );
}
```

Mouse Event

Because most of the actions are performed using the mouse, the MouseEvent events are enhanced with some extra properties and methods:

- `x` and `y`: mouse coordinates relative to the canvas element (top-left)
- `dragging`: true if the user was dragging the mouse (one button clicked).
- `canvasx` and `canvasy`: mouse position in webgl viewport coordinates (bottom-left).
- `deltax` and `deltay`: amount dragged in both axis.
- `leftButton`,`middleButton` and `rightButton`: if the button is pressed
- `dragging`: true if any button is pressed.

Reading the current Input state

Sometimes you want to read the user raw input like the mouse coordinates, if a keyboard key is pressed or the gamepad axis.

In these situations you can use the `LS.Input` class to see the input state of:

- `LS.Input.Mouse` to read the mouse state
- `LS.Input.Keyboard`to read the keyboard state
- `LS.Input.Gamepads` to read the gamepads state

Mouse Input

When reading the mouse input you can access the `LS.Input.Mouse`.

To access the mouse coordinates there is several ways:

- `x`: x coordinate in canvas coordinates
- `y`: y coordinate in canvas coordinates (0 is the bottom canvas position)
- `clientx`: x coordinate in client coordinates
- `clienty`: y coordinate in client coordinates (0 is the top canvas position)

To get the mouse buttons state you can read the `LS.Input.Mouse.buttons` and mask to read every button state:

```
if( LS.Input.Mouse.buttons | GL.LEFT_MOUSE_BUTTON )
    //...
```

or use the `left_button`,`middle_button` and `right_button` flags to check the state.

```
if( LS.Input.Mouse.left_button )
    //...
```

Touch events

By default LiteScene converts every touch event in a mouse event, this disables using multitouch gestures but we will work on this in the future.

Keyboard

To read the keyboard state you can use the `LS.Input.Keyboard` where you have all the keys state:

`LS.Input.Keyboard["UP"]` will tell you if the cursor UP key is pressed.
`LS.Input.Keyboard["a"]` will tell you if the character 'a' key is pressed.

or you can use the keycode:
`LS.Input.Keyboard[32]` will tell you if the RETURN key is pressed.

Gamepads

To read the gamepads state you use a similar approach:

```
var gamepad = LS.Input.Gamepads[0];
if(gamepad) //is gamepad connected
{
  var x = gamepad.axes[0]; //here axes must be referenced using a number
  if( gamepad.buttons[0].pressed )
  {...}
}
```

or to read the axes directly:

```
var x_axis = LS.Input.getGamepadAxis(0,"LX"); //returns the LEFT AXIS value of the gamepad 0
```

or to read the buttons directly:

```
if( LS.Input.isGamepadButtonPressed(0,"A") ) //returns if the "A" button of the gamepad 0 is pressed
  ...
```

Example

This example moves the object using the gamepad or the keyboard cursors.

```
//@unnamed
//defined: component, node, scene, globals

this.moving_speed = 100;
this.rotating_speed = 90;

this.onStart = function()
{
}

this.onUpdate = function(dt)
{
  var x_axis = 0;
  var y_axis = 0;

  var gamepad = LS.Input.getGamepad(0);
  if(gamepad)
  {
    x_axis = LS.Input.getGamepadAxis(0,"LX");
    y_axis = LS.Input.getGamepadAxis(0,"LY");
  }

  if(LS.Input.Keyboard["UP"])
    y_axis -= 1;
  if(LS.Input.Keyboard["DOWN"])
    y_axis += 1;
  if(LS.Input.Keyboard["LEFT"])
    x_axis -= 1;
  if(LS.Input.Keyboard["RIGHT"])
    x_axis += 1;

  x_axis = Math.clamp(x_axis,-1,1);
  y_axis = Math.clamp(y_axis,-1,1);

  node.transform.translate(0,0,y_axis * dt * this.moving_speed);
  node.transform.rotateY(-x_axis * dt * this.rotating_speed);

    node.scene.refresh();
}

this.onGamepadConnected = function()
{
  console.log("Gamepad connected!");
}
```

Operating with Vectors, Matrices and Quaternions

When scripting any 3D application usually you will need to do mathematic operations like adding or multiplying vectors, normalizing, doing the cross or dot product, transforming vectors by matrices, rotating quaternions, etc.

Instead of providing out own mathematical library, LiteScene uses gl-matrix as the base library for geometric operations.

Created by Brandon Jones, this library has proven to be very powerful and bug-free, but has a particula syntax that any user must understand in order to use it.

Understanding gl-matrix

gl-matrix doesnt create classes to store vectors, matrices and quaternions. Instead it relies in the Float32Array class.

This means that any vector, matrix or quaternion created using gl-matrix is stored in a regular Float32Array(...).

Because of this the performance is very good and when storing lots of vectors in a single array, the array can be a regular typed array.

But this also means that we cannot rely in the classical syntax of `myvector.normalize()`, instead we must call `vec3.normalize( myvector, myvector )`.

Also, gl-matrix allows to apply any operation over an existing vector without creating new ones, this way it reduces the garbage.

Creating a Vector3

To create a vector we have several ways:

```
var myvector = vec3.create(); //default 0,0,0
var myvector = vec3.fromValues(x,y,z);

var myvector = vec3.clone([x,y,z]);
var myvector = new Float32Array([x,y,z]);
```

You can also create a vec2 or vec4.

Operating with vectors (vec2,vec3,vec4)

When operating over any container you must pass the output container as the first parameter and the result will always be the first parameter.
And you dont have to worry if the output container is the same as one of the inputs, gl-matrix handles that for you to avoid strange errors.
This way we can control how much garbage is generated in our code:

```
var result = vec3.add( vec3.create(), v1, v2 );
```

Here is a list of some of the operations available:

- `vec3.add( out, a, b )`: add two vec3
- `vec3.sub( out, a, b )`: substracts two vec3
- `vec3.mul( out, a, b )`: multiply two vec3
- `vec3.div( out, a, b )`: divide two vec3
- `vec3.scale( out, a, f )`: multiply a vec3 by a scalar
- `vec3.normalize( out, a )`: normalize a vector
- `vec3.cross( out, a, b )`: cross product between two vectors
- `vec3.dot( out, a, b )`: dot product between two vectors
- `vec3.length( a )`: length of the vector
- `vec3.clone( a )`: returns another vector
- `vec3.copy( r, a )`: copies a vec3 to the output

To have a full list of vec3 operations the gl-matrix vec3 documentation

Operating with matrices (mat3,mat4)

For matrices the class is mat3 and mat4.

To have a full list of mat4 operations the gl-matrix mat4 documentation

Operating with quaternions (quat)

For quaternions the class is called quat.

To have a full list of quat operations the gl-matrix quat documentation

Using the LS.Player

The LS.Player is the class used to embed and launch an scene inside a website so you do not need to use WebGLStudio to visualize it.

The Player is in charge of several tasks:

- Set up the render context
- Load the scene and the scripts associated to it
- Show progress bar during loading (useful in big scenes)
- Start playing the scene once everything is loaded
- Process the main loop to call the render and update methods
- Capture the input and send it to the specific callbacks
- Handle the GUI elements so they overlap properly with the scene

And because it is just one single class it is very easy to create a website that launches one scene:

Usage

If you want to play an scene in your browser, here are the basic steps:

Include the libraries and dependencies:

```
<script src="external/gl-matrix-min.js"></script>
<script src="external/litegl.min.js"></script>
<script src="external/Canvas2DtoWebGL.js"></script>
<script src="js/litescene.js"></script>
```

Create the player:

```
var player = new LS.Player({
    width:800, height:600,
    resources: "resources/",
    shaders: "data/shaders.xml"
});
```

Attach to Canvas to the DOM:

```
document.getElementById("mycontainer").appendChild( player.canvas )
```

or you can pass the canvas in the player settings as `{ canvas: my_canvas_element }`

Load the scene and play it:

```
player.loadScene("scene.json");
```

Some additional options you can pass to the player:

- **canvas**: the canvas element where to attach the render context (string selector or element)
- **width** and **height**: if you want the player to create the canvas by itself
- **container**: which DOM element where to attach the canvas, if not specified then the body is used.
- **container_id**: the container id if no container element is supplied.
- **loadingbar**: if true it will show the loading bar (default false)
- **redraw**: if false the scene wont be redraw constantly
- **autoresize**: if true the canvas will always try to match the parentNode size
- **autoplay**: if false the player wont play the scene unless you do it manually
- **shaders**: the path to the shaders.xml file, if omited then "data/shaders.xml"

You can also add to the options information for the WebGLContext:

- **alpha**: true if you want the canvas to be transparent
- **premultipliedAlpha**: if you want to multiply the alpha by the color

In case you want to overwrite the loading bar gizmo shown while loading the scene, you must overwrite the player.renderLoadingBar:

```
player.renderLoadingBar = function( loading_info )
{
    //to know how much of the main scene file has been loaded: loading_info.scene_loaded
    //to know how much of the resource files as been loaded: loading_info.resources_loaded
}
```

Post-processing and Camera FX

It is common to apply post-processing FX to the image rendered to give it a final touch.

LiteScene provides some Components that help apply those efects, but it also provides an API that makes very easy to create your own FX manually.

Camera or Global Effects

There is one thing to keep in mind when creating an FX: if you want it applied to a single camera viewport or to the final frame.
If you want to apply it to the camera then the FX should be binded to the camera, otherwise if should be binded to the whole scene.

RenderFrameContext

To apply any FX to an image first we need to render the scene to a texture, and then apply the effect to the content of that texture.

But creating and setting the texture to render could be a complex process, also you need to keep in mind several situations:

- Which size should the texture be? That would depend on the viewport size which is affected by the window size and the camera viewport.
- Which type should the texture be? Maybe we can render to a low-precision texture, but in some cases we may need high precision.
- What if we want to store the depth buffer also in a texture to use it in our effects?
- What if I want to render to several textures at the same time? GPUs allow to output to several texture to have extra info per pixel.
- How to render the final buffer to the screen?

To help tackling those situations LiteScene provides a special class called LS.RenderFrameContext. Here is a list of the properties:

- **width** and **height**: this defines the final size of the texture, if width or height is 0 then the respective size of the viewport will be used. If we want to downscale the viewport size we can use negative numbers (-1 means half, -2 a quater, etc).
- **precision**: here you can specify the data type of the texture, values are:
- **RenderFrameContext.DEFAULT_PRECISION**: gl.UNSIGNED_BYTE (alghough it could be changed with the var RenderFrameContext.DEFAULT_PRECISION_WEBGL_TYPE)
- **RenderFrameContext.LOW_PRECISION**: gl.UNSIGNED_BYTE Good for regular renderings
- **RenderFrameContext.MEDIUM_PRECISION**: gl.HALF_FLOAT_OES if supported, otherwise gl.FLOAT, otherwise, gl.UNSIGNED_BYTE (good for renders which very bright spots that need to preserve them after FX)
- **RenderFrameContext.HIGH_PRECISION**: gl.FLOAT if supported (otherwise gl.UNSIGNED_BYTE) Good for deferred (slow)
- **filter_texture**: in case you want to set the magFilter of the final color_texture to NEAREST, so it looks pixelated.
- **adjust_aspect**: in case the frame has an aspect ratio of the final viewport, this forces the renderer to correct it.
- **use_depth_texture**: allows to store the depth in another texture so it can be used in the effects.
- **num_extra_textures**: number of additional textures to bind as render buffer

To enable a RenderFrameContext just call `enable`, and when finished call `disable`. To show it on the screen call `show` although if you want to apply any FX you can read the textures using the `getColorTexture` method.

FXGraphComponent

Another way to apply FX is using the graph system. This way is much more intuitive but it consumes way more memory (because every graph node uses its own texture).
Just connect the "Rendered Frame" graph node to any gltexture node and the final output to viewport.

Using a Script

When using scripts you can easily create your own effects, here is an example:

```
//@SimpleCameraBlurFX

this.high_precision = false;
this.iterations = 4;

var frame = new LS.RenderFrameContext();
frame.width = 0;
frame.height = 0;
frame.precision = LS.RenderFrameContext.LOW_PRECISION;
frame.filter_texture = true;

this.onEnableFrameContext = function()
{
    frame.precision = this.high_precision ? LS.RenderFrameContext.MEDIUM_PRECISION : LS.RenderFrameC
  frame.enable();
}

this.onShowFrameContext = function()
{
    frame.disable();
  var tex = frame.getColorTexture();

  var tmp = GL.Texture.getTemporary( tex.width, tex.height, { type: tex.type  } );
  var tmp2 = GL.Texture.getTemporary( tex.width, tex.height, { type: tex.type  } );

  tex.copyTo( tmp );

  var iterations = Math.clamp( this.iterations, 0, 20 );
  for(var i = 0; i < iterations; ++i )
    tmp.applyBlur(1<<i,1<<i,1,tmp2);

  tmp.toViewport();

  GL.Texture.releaseTemporary(tmp);
  GL.Texture.releaseTemporary(tmp2);

}
```

TextureFX

Also LiteScene provides a nice and fast way to apply several common effects to a texture. You need to use the LS.TextureFX class.

This class allows to concatenate different basic effects (like vigneting, brightness and contrast, edge detection) and pack all the efects in one single shader automatically.

check the TextureFX class for more info.

Prefabs

It is common when creating scenes to reuse the same objects in the same scene or between different scenes.

To avoid having to create the object again LiteScene allow to create prefabs.

A prefab is a description of a SceneNode (its properties, components and children).

By creating a Prefab you can store that node info as a resource and invoke it whenever you need it.

Creating Prefabs

You can create prefabs by code or using the WebGLStudio editor.

Programming new components

LiteScene is a component based engine.

That means that any action performed by the system, like rendering on the screen, adding lights, applying postprocessing effects, or executing scripts, comes from one of the many components.

Components are attached to the nodes in the scene to extend the properties of every node, this way the system is very modular and easy to extend.

Although the system comes with many components sometimes you need special features. In that case you have two choices, to create a Script or to code the Component. Scripts are heavier and offer less freedom that regular components and they are not as easy to add as default components.

Check the Scripting guide to know how to create Script components and which are their benefits.

If you dont want to rely on Scripts you can create your own components and add them to the components's pool.

Creating a new Component class

Creating a component is easy, which just a few lines of code you will have your component ready to use, the complex part is ensure that this component is going to interact propertly with the system, here is a list of common steps that you will have to fulfill to create a component:

- Create the **component class** and register it in the system
- Add the **configure** and **serialize** methods in case you want to perform special actions when storing/restoring the state
- Add the `onAddedToScene` and `onRemovedFromScene` methods to hook the appropiate events with the system
- Define the type or widget information for the public vars, this helps the editor show propper interfaces. Or in the last resort, define the full interface of the component.
- Be sure that any project that uses this component is including the code of this component as an external or global script.

The file containing the code of new components can be included in the HTML, or in the scene using the external_scripts or global_scripts.

The component class

Lets start by creating a javascript file with a very basic component class.

```
function MyComponent( o )
{
    this.myvar = 1;

    if(o)
        this.configure(o);
}
```

As you can see the component will receive an object as a parameter, this is in case we want the object to restore a previous state, when thats the case the object will contain all the serialized data. For now we just pass that object to the configure method.

Base properties

Every component has some important properties and methods by default. They are added to the prototype of the class when it is registered. Here is a list of the properties:

- `root`: the node where the component is attached (direct access: `_root`).
- `scene`: the scene where the root node belongs to (direct access: `_in_tree`).
- `uid`: the unique id string, it is autogenerated when attached in case it doesnt have one. (direct access: `_uid`).

Check the Component class to see all the methods.

Adding public variables

Every property of the class is public and can be accessed. If you are working from the WebGLStudio editor then you will see that all properties are exposed in the component interface. To avoid that you should start all the properties with an underscore (`this._internal_data = 10;`) and only use regular names with variables that can be edited.

When creating variables that are editable remember that only some basic types are supported. If the variable has a special type then the editor wont know how to create an interface.

If you want a variable to have an specific editor you can specify it like this:

```
MyComponent["@mydirection"] = { type: "enum", values: ["north","south","east","south"] };
MyComponent["@profile_pic"] = { type: "resource" };
```

Or if you want to have an specific widget:

```
MyComponent["@age"] = { widget: "slider", min:1, max:90, step:1, precision:0 };
MyComponent["@gender"] = { widget: "combo", values: ["male","female","other"] };
```

If you are planning to create a custom interface for this component, check the guide for custom editor interfaces

Registering the component

The next step is to register the component in the system so it can be listed in the components list in the editor.

```
LS.registerComponent( MyComponent );
```

This function is not only registering the component but also adding the mandatory methods to the component that you didn't fill (like serialize, configure).

When a component is registered it checks if the current scene contains this component, and if that is the case it replaces it.

Adding behaviour

We have our component but its not doing anything special so we need to add some behaviour.
The best way to add behaviour to a component is binding some functions to specific events triggered by the system (like the render, mouseMove or update events).

When we want to bind an action there are several ways but the most common one is:

```
  LEvent.bind( scene, "update", this.onUpdate, this );
```

Where the first parameter is the instance that will trigger the event, the second the name of the event, the thirth the callback and
the fourth parameter is the instance where you want to execute the callback. You could use mycallback.bind(this) and skip the fourth parameter but the problem with that approach is that you wont be able to unbind the event in the future (because the Function.bind returns a different function every time), so try to pass always the fourth parameter.

If we want to unbind the event we could call unbind for every event or directly unbindAll to the specific instance:

```
  LEvent.unbindAll( scene, this );
```

When binding events there is an important restriction: **you have to be sure that once this node is no longer attached to the scene (because the node has been removed) your component is no longer doing any action.**.

So when attaching events the best way to do it is using the **onAddedToScene** method in your component:

```
MyComponent.prototype.onAddedToScene = function( scene )
{
  LEvent.bind( scene, "update", this.onUpdate, this );
}
```

And for the opposite, create the method **onRemovedFromScene**:

```
MyComponent.prototype.onRemovedFromScene = function( scene )
{
  LEvent.unbind( scene, "update", this.onUpdate, this );
}
```

If your actions are more related to the node then use the onAddedToNode and onRemovedFromNode.

If you want a list of events follow the the LS Events list

Serializing

Components usually store data that the user can change to control the behaviour, and that data must be persistent so changes done in the scene will be saved.

The system will call the `serialize` method of every component to extract the state (if you don't want that this component is persistent you must the `component.skip_serialize = true;`).
And when the component state is restored it will call the method `configure` (or pass it as a first parameter in the constructor).

If the user doesn't define a configure or serialize method the system will create one by default.
This default methods will save any public variable of the component (the ones that doesnt start with an underscore character) and restore its state.

Keep in mind that sometimes you want to do some computations when the component state is retrieved or when the component state is assigned, so define your methods if that is the case.

```
MyComponent.prototype.serialize = function()
{
    return {
        myvar: this.myvar
    };
}

MyComponent.prototype.configure = function(o)
{
    if(o.myvar !== undefined) //we can control if the parameter exist
        this.myvar = o.myvar;
}
```

Special Events and Actions

Some components could trigger events (p.e. the user has clicked the node) or could be connected to actions (it must play an animation).

To let the editor know which special events and actions can perform a component, you must specify it:

```
//returns which events can trigger this component
MyComponent.prototype.getEvents = function()
{
    return { "start_animation": "event", "end_animation": "event" };
}

//returns which actions can be triggered in this component
MyComponent.prototype.getEventActions = function()
{
    return { "play": "function","pause": "function","stop": "function" };
}
```

Testing your component in the editor

The problem with our component is that is stored in a javascript file that our editor doesnt load, so there are different ways to force the editor to load new files.

In our case what we are going to do is add the file the external scripts in the Scene settings, to do so go to Scene - Settings and add the URL of the file in the external scripts.
Once the file is loaded you will see your component when clicking the [add Component] button in any node.
If you add the component to a node you also will see that the system has detected that your component has a local variable (this.myvar)
and has created a tiny interface so you can modify its value.

Example

Here is a basic component that makes the node rotate:

```
function RotateComponent( o )
{
  //define a public variable
  this.speed = 10;

  //call configure if the components recibe a state in the constructor
  if(o)
    this.configure(o)
}

//specify the widget to show on the editor, otherwise an automatic one will be created
RotateComponent["@speed"] = { widget:"slider", min:-10, max:10 };

//we bind the events once this component gets attached to a scene
RotateComponent.prototype.onAddedToScene = function(scene)
{
    LEvent.bind( scene, "update", this.onUpdate, this );
}

//and remove the events once it is detached
RotateComponent.prototype.onRemovedFromScene = function(scene)
{
    LEvent.unbindAll( scene, this );
}

//This will be called every time the update method is launched because we bound it that way
RotateComponent.prototype.onUpdate = function(e,dt)
{
  this._root.transform.rotateY(dt*this.speed);
}

//in this case we dont need to do a serialize method but just to show how it is done
RotateComponent.prototype.serialize = function()
{
    return {
      speed: this.speed
     };
}

RotateComponent.prototype.configure = function(o)
{
  if(o.speed !== undefined) //we can control if the parameter exist
    this.speed = o.speed;
}

//Add this component to the components pool in the system
LS.registerComponent( RotateComponent );
```

The Render Pipeline

The rendering pipeline of LiteScene has several steps that are important to understand if you want to modify how it behaves.
The reason to have some many steps is to ensure all the cases are considered.
If you are happy with the current rendering pipeline you do not need to read this article.

WebGL, LiteGL and LiteScene

The first and most important part is to understand the layers involved in the rendering process.

Obviously in the lowest point we have the WebGL API supplied by the browser. You can call the WebGL API directly if you want, just keep in mind that during the rendering of a frame the Renderer will assume that the state of the API is in the state itw as left by it, so be careful when changing the state.
But I recommend not to call WebGL directly an use LiteGL instead.

LiteGL is a low-level wrapper of WebGL that makes it easy to compile shaders, create meshes or upload textures, without performance loss.
It is important to know how to use it if you plan to modify the rendering pipeline. Although you are more prone to interact with LiteScene than with LiteGL.

LiteScene rendering methods are in charge of creating the final frame, and those are the ones that you will have to tweak to change how the render pipeline works.

LS.RenderInstance

First we need to understand the atomic class to render stuff that uses LiteScene, it is called RenderInstance.

A RenderInstance represents one object to render, and it contains the mesh, the material and the flags that must be taken into account when rendering that mesh.

So every component of the scene that plans to render something on the screen needs to create a RenderInstance and supplied to the system when the collectRenderInstances event is generated.

Take into account that a RenderInstance is not a low-level API render call, because it still depends in many parameters to determine how to render it.

For example, the same RenderInstasnce will be rendered with different shaders depending if it is being rendered to the color buffer or to the shadow buffer.

Also one SceneNode can generate one RenderInstance, several or none, it depends on what it wants to do.

The Renderer

Most of the rendering pipeline is contained in the LS.Renderer static class.
This class is in charge of taking a scene and generate the final frame.
It is important to take into account that to generate the final frame sometimes it is required to render the scene several times (for shadowmaps, reflections, secondary cameras, ...).
So if you plan to change the rendering pipeline maybe is better to change only the parts that really matter.

The Renderer class is used by many components to do intermediate steps, so do not replace it completly unless the new class contains most of its methods.
The Renderer also keeps tracks of the current state of the rendering in process, so components can retrieve info during the rendering (stuff like the current camera, active samplers, lights, etc).

Renderer.render

This is the most important method from the Renderer, is the one that generates the final frame.
Here is a list of the steps performed by the render pipeline when calling the render function:

1. **Collect visible data**
2. **processVisibleData** which will call **scene.collectData** to collect all visible data
3. prepare RenderInstances (compute rendering order)
4. prepare Lights (generate shadowmaps)
5. prepare Cameras (they are sorted so cameras that are rendered to texture are renderer first)
6. Trigger several events to generate intermediate content (reflections)
7. In case there is a global framebuffer, enable that (to render the final frame to a texture and apply an FX)
8. **renderFrameCameras**: For every active camera
9. Trigger camera events in case this camera requires an special buffer
10. **renderFrame**: Render Scene from Camera view point
    1. **enableCamera** assign viewport and matrices according to camera
    2. **sortRenderInstances** rearrange the render instances so they are rendered in the propper order (opaque first, blend last, and taking

into account priority)
3. clear Buffer (or not)
4. **renderInstances** iterate through every RenderInstance and call to renderColorPassInstance or renderShadowPassInstance

11. apply FX to this buffer

Most of the rendering calls are performed from the **renderColorPassInstance** so check the chapter about that function to understand better how a single RenderInstance is rendered.

Careful, if you want to issue any special rendering pass during the render of a frame, **you must never call LS.Renderer.render from an event dispatched by the rendering**, this will create a recursive loop. You can bind to `"beforeRenderMainPass"` event and call functions like `renderFrame` to render the view from one camera, or do it manually calling `enableCamera` and `renderInstances`.

Render Events

In case we want to bind some events during the render here is the order at which they are triggered and the name of the callback that you can define in a script to catch it:

- **"beforeRender"** (script: onSceneRender) Just after launching the `LS.Renderer.render(...)`
- **"collectRenderInstances"** (script: onCollectRenderInstances) When collecting resources to show in the scene
- **"enableFrameContext"** (script: onEnableFrameContext) To enable to which LS.RenderFrameContext you want to do the render
- **"beforeCameraEnabled"** In case we want to change any parameter of the camera when rendering
- **"beforeRenderInstances"** (script: onRender) before rendering every instance of the scene
- **"afterRenderInstances"** (script: onAfterRender) after rendering every instance of the scene
- **"showFrameContext"** (script: onShowFrameContext) when you want to show the context
- **"renderHelpers"** (script: onRenderHelpers) to render helper objects (gizmos, grids, etc)
- **"renderGUI"** (script: onRenderGUI) to render 2D info in the immediate mode
- **"afterRender"** (script: onAfterSceneRender) after all the content of the scene has been rendered.

Collecting the data

The scene contains many cameras, lights, render instances and other items that could affect the final frame, so we need to have all the data stored in the propper containers.

This methods try to extract all the useful information from the scene and prepare it so it is ready to be used during the rendering.

It is important to notice that once this action is performed and while rendering the frame, any changes applied to the scene wont be reflected till the next frame.

Materials

When rendering a single instance the actions that must be performed depend on the kind of material it has assigned.
Different materials can specify different ways to render an instance, but to understand better the rendering pipeline lets focus in two different materials.
The ShaderMaterial and the StandardMaterial

ShaderMaterial rendering

This is the simplest one, when rendering an instance that has a ShaderMaterial applied to it then Renderer will call the render function of the material.
The ShaderMaterial finds the Shader, passes the uniforms, and calls the render method in the RenderInstance.

This is the most straight forward shader, but it has some limitations. Because the shader assumes a fixed set of parameters, when rendering this instance it wont be affected by the surroundings.

This means that it won't have shadows or be affected by the scene lights or get any modifiers applied to it (like Skinning or Morphing) unless the shader specifies it.

But it is the one that has the best performance.

StandardMaterial rendering

Sometimes we don't want to take care of the shader, we just want to specify some properties and let the render pipeline decide which is the best shader to apply.

In those situations the system has to be aware of the different modifiers to apply to the shader based in all the actors in the scene, like:

1. lights: because lights could have different type, or have shadowmaps, or projector textures, or special shaders.
2. nodes: because nodes can have deformers applied to them (skinning, morph targets)
3. scene: because maybe there is a clipping plane
4. renderer: because maybe the renderer is using an special pipeline

All those actors can affect the shader, changing its behaviour. So an StandardMaterial cannot have an specific shader applied to it.

Instead, the RenderPipeline computes the shader based on all those actors and renders the RenderInstance with the final shader.

This process is slower than using a fixed shader but ensures that people with no knowledge about shader coding can create its own materials easily.

How does it achieve this? Well, every actor creates something called a ShaderQuery that stores all the information that must be taken into account when computing the shader of the render instance, and when rendering the ShaderQuery returns a shader that matches all those properties.

Multi light rendering

When rendering an scene we want to be sure than an object can be affected by multiple lights. This is achieved by using a multi pass rendering approach.

This means that for every light affecint the RenderInstance, the pipeline is going to render that instance. This could lead to bad performance when we have an scene with several meshes an lights.

RenderState

Materials can control the way they are rendered by changing the flags in the GPU during rendering.

This way a material can decide if it is z-culled, two-sided, blended, etc. The info about how it should be rendered is contained in the `LS.RenderState` class in the `material.render_state` property.

Post-processing Effects

A camera could be rendered to the screen or to a texture, in which case the texture could have FX applied to it.

To render to a texture we use a class called RenderFrameContext, which helps setting up the context.

There are several components that allow to apply FX to the camera, just keep in mind that we could apply an FX per camera or to the whole scene.

Those components will bind events to the camera enableFrameBuffer (or the scene enableFrameBuffer), so they can redirect the render to the

Resources

When working with LiteScene you will need to retrieve many resources to use during the render or to control the behaviour of the application.

Classes

Resources can be of many types, depending on the info they store:

- **Mesh** to store the geometry of the objects we are going to render on the scene. This class is defined in LiteGL.
- **Texture** to store the images uploaded to the GPU, this class is defined in LiteGL.
- **Prefab** to store fragments of the scene that could be instantiated many times in the scene.
- **Animation** to store tracks containing keyframes over a timeline for a property.
- **ShaderCode** to store and parse GLSL code used by ShaderMaterial.
- **Resource** Generic class to store text content. Used to store javascript files, data, html, etc.
- **Material** to store the properties of a material of any class.
- **Pack** which contains several resources in one single file (useful for deploy).

Fullpath

Every single resource should have a string to identify it in the system, this string is usually the filename or the url to fetch the file.

In case a resource should not be loaded or stored in the server (a local resource) the name should start with the character colon ':'.

To access the fullpath of any resource you can get if using the property fullpath.

Although some resources could have also the property filename (a file without a folder yet) or remotepath (the name in the server), the important thing to take into account is that all should have a fullpath.

ResourcesManager

There is a global class called the `LS.ResourcesManager` (also abreviated as `LS.RM`) which is in charge of loading, processing and storing all resources available by the system.

This way we can ensure that we are not loading a resource twice, or bind events when new resources are loaded.

There are many methods in the `LS.ResourcesManager` class, so check the documentation carefully to understand all its features.

Here is a list of the common methods:

- `load( fullpath, options, on_complete)` used to ask the ResourcesManager to load a resource, you can pass a list of options of how the file should be processed once is loaded, and a final callback(resource, url) to call once the file has been loaded. If the file is already loaded it wont load it again.
- `getResource( fullpath )` to retrieve a resource, if not found null is returned.
- `registerResource( filename, resource )` to make a resource available to the system
- `unregisterResource( resource )` to remove a resource from the system

Resources are stored in a container called `LS.ResourcesManager.resources` but also there are independent containers for textures and meshes to speed up fetching.

Paths

When loading resources we could need to fetch the files using a root folder as a base path, this way resources do not need to have absolute paths.

The paths where the resources will be loaded is specified using the `LS.ResourcesManager.setPath` and stored in the `path` property.

To avoid cross-site origin scripting problems, the ResourcesManager allows to specify a path that will be used as a root path when fetching remote files, it is stored in the `proxy` property.

Formats

Every resource must be loaded and parsed, and depending on the type this process could differ (like if it is a text file or a binary file).

All that information is controlled by the LS.Formats class, that contains information about every fileformat supported by the system.

To know more about File Formats check the File Formats guide

Example

```
this.onStart = function()
{
   var that = this;
   LS.ResourcesManager.load("data/myfile.txt", function(data) {
      that.processData( data );
   });
}

this.processData = function( data )
{
  //...
}
```

Scene

To familiarize with LiteScene first you need to understand how the scene is composed.

The LS.SceneNode

A LS.SceneNode represents an object in your scene (similar to how a GameObject works in Unity).

It can have a name, a transform to define where it is, and a list of components to add behaviour and visual properties.

Everything visible or that react to our scene must be inside a LS.SceneNode.

Names and UIDs

Every node can have a name in a String form, this name doesnt have to be unique.

Besides the name, every node has a UID (unique identifier) that tries to be as universal as possible to avoid collisions when loading nodes from several scenes.

UIDs are created using random numbers, incremental numbers and timestamps so they should be very unique.

```
node.name = "Ball";
console.log( node.uid ); //will show something similar to: "@NODE--72553e-27b-1a284aa-5"
```

Inserting nodes inside nodes

SceneNodes belongs to a tree like structure, where every node can have other nodes inside, to do this we can attach a node to another node:

```
node.addChild( other_node ); //adds other node inside node
```

Child nodes are stored in `node.children` and the parent node of a node is stored in `node.parentNode`.
Check the documentation for more info about how to attach or remove nodes from a node.

Layers

Every node belongs to several layers (or none), this helps filter which object should react or be visible with our scene.

Layers are stored using a number where every bit represent to which layer belongs (to a maxium of 32). The layers names are stored in the SceneTree.

```
node.layers |= 1; //adds the layer 1 to this object
```

SceneTree

The SceneTree is the global container for the whole scene.

To acess the current active scene you can use `LS.GlobalScene`

This contains a root SceneNode where all the nodes of the scene are located.

To access the root node you can go to `LS.GlobalScene.root`

Scene information

You can access several information from the Scene like:

- **time**: the time the scene has been running in seconds (this value only increases if it is in play mode).
- **global_time**: the system time in seconds (similar to getTime()).
- **frame**: the current frame number (how many times the function `LS.Renderer.render` has been rendered with this scene).
- **layer_names**: an array containing the name for every layer.

Documentation

For a more detailed description check the documentation.

Scripting

LiteScene allows to run scripts so users can code the behaviour of the application from with in the editor itself.

The info about the scripts can be stored inside the scene so when a new scene is loaded it loads its scripts too.

There are several ways to interact programmatically with LiteScene, every method is better suited for different purposes.

Before delving into the Script component keep in mind that you can create your own components without the need of using the Script component. To know more about it check the programming components guide.

Using Script components

The Script component is the easiest way to add behaviour to a project. You can attach a `Script` or a `ScriptFromFile` component to any node. The difference between both is that `Script` stores the code inside the component while `ScriptFromFile` references the code from a resource file (better suited when sharing the same behaviour among different nodes or projects).

`ScriptFromFile` behave as regular `Script` but because its load is asynchromous it means their context will be created later in time after the scene has started, keep that in mind (events like start will me called once the node is loaded).

The script context

Every script has its own execution context usually referred as the script context.

The context is created when the component is configured and the code loaded.

When programming inside a `Script`, the context is the `this` of your code:

```
this.foo = 100; //adds a property foo to the script context
```

To access the context of another script component just access the context property throught the component:

```
var node = scene.getNode("mynode");
var script_component = node.getComponent( LS.Script ); //or LS.ScriptFromFile, depending on the compo
script_component.context.foo = 10;
```

Check the section below to know other ways to access a context of another component.

Global vars of every script

Besides all the objects of the system, every script has five globals vars associated to that script:

- **scene**: the scene where the node of this script component is attached, (usually the same as `LS.GlobalScene`)
- **node**: the node where this script component is attached, (equivalent to `component._root`)
- **component**: the script component itself
- **transform**: the transform of the node where the script is attached
- **globals**: the container shared among all the Scripts, the same as `LS.Globals`

So feel free to access them from inside your script at any time.

Local vars and functions

Every `var` (or `function`) defined in that scope is local to the context so it cannot be accesed from outside of the scope.

Unless we make it public or we make a setter/getter.

Public vars

If the user wants to make local vars or methods accesible from other scripts, graphs or animation tracks (or the editor), they need to be made public, to do so they must be attached to the context itself:

```
this.number = 1; //this var will be public
```

Sometimes may be helpful to specify the type of the var to the system, this way the var can be properly connected using Graphs, or animated using Animation Tracks.

In that case the user can use:

```
//to create the var
this.createProperty("myvar", [1,1,1], LS.TYPES.VEC3 );

//to access  it
this.myvar = [10,10,10];
```

Specifying types is important when the types are not basic types, and if you are using WebGLStudio the system will create appropiate widgets to interact with them.

Also, when using WebGLStudio, there is also the option to specify widget properties to have a better UI for this script:

```
this.createProperty("myvar", 0, {type: "number", widget:"slider", min:0, max:100, step:1});
this.createProperty("myvar2", "yes", {type: "string", widget:"combo", values:["yes","no","maybe"]});
```

Or to create even Arrays that can be edited through the editor:

```
//in this case is an array of textures but you can ignore the type if is an array of anything
this.createProperty( "myarray", [], { type: "array", data_type: LS.TYPES.TEXTURE });
```

Or if you want to have a handy button in the component interface when using the editor:

```
this.createAction( "Click me", this.myCallback );
```

Events

To interact with the system, scripts need to attach callbacks to events dispatched by the different elements of the scene, mostly by the scene (but could be the `LS.Renderer`, the `LS.ResourcesManager`, etc).
For a better list of execution events check the events guide or if you want events from a component check the specific component documentation.
To bind an event you can call the bind method:

```
this.bind( scene, "update", myfunction );
```

Keep in mind that myfunction must be a public method attached to the context (p.e. `this.myfunction`), otherwise the system wont be able to remove it automatically.

API exported methods

However, there are some events that scripts usually want to use, like **start**, **init**, **render**, **update** and **finish**.
You do not need to bind (or unbind) those events, the Script component does it automatically when it detects a method in the context with an specific name (depending on the event):

```
this.onUpdate = function(dt) { ... };
```

Here is a list of the automatically binded events:

- **onStart**: triggered by scene "start" event, remember that if your script is created after the scene starting you wont receive this.
- **onFinish**: triggered by scene "finish" event, used in the editor when the user stops the play mode.
- **onPrefabReady**: triggered by the node "prefabReady", used to access components or node that come from the prefab
- **onUpdate**: triggered by scene "update" event. it receives the delta time in seconds.
- **onClicked**: triggered by the node "clicked" event. Remember that you need an `InteractiveController` in the scene to dispatch this events.
- **onCollectRenderInstances**: triggered by node "collectRenderInstances" event. To pass RenderInstasnces
- **onSceneRender**: triggered by scene "beforeRender" event. Used to prepare stuff before any rendering is done.
- **onRender**: triggered by the node "beforeRenderInstances" event. Used to direct render stuff before the RenderInstances are rendered.
- **onAfterRender**: triggered by the node "afterRenderInstances" event. Used to direct render stuff after the RenderInstances are rendered.
- **onRenderHelpers**: triggered by scene "renderHelpers" event. To direct render stuff related to the editor.
- **onRenderGUI**: triggered by scene "renderGUI", to render stuff in 2D (using the canvas2D).
- **onEnableFrameContext**: triggered by the scene "enableFrameContext" event. Before rendering the final frame, used to setup a special RenderFrameContext and apply FX to the final image.
- **onShowFrameContext**: triggered by the scene "showFrameContext" event. After the final frame, to show the frame into the viewport.
- **onRemovedFromScene**: called when the node where the script belongs is detached from the scene.
- **onGetResources**: called when the script needs to collect resources. This function receives an object that needs to be filled with the fullpath : type of the resources it uses so they can be automatically loaded when the scene is loaded.
- **onFileDrop**: called if the user drags and drop a file over the scene. Receives and object that contains the file.

Keep in mind that you are free to bind to any events of the system that you want. Just remember to unbind them from the `onRemovedFromScene` so no lose binds are left.

Input

You can bind events for actions performed by the user (like mousedown, keydown, etc) or read the input system directly (using the LS.Input object).

Check the Input guide to see more information abour reading the input.

Serialization

Any data attached to the context whose name doesn't start with the character underscore "_" will be serialized automatically when storing the scene and restored when the context is created. Keep in mind that when serializing any property it is stored as a base type, so avoid setting public variables of special classes, only store properties of the common types like String, Number, Bool, or Arrays of basic types.

If you want to store stuff that shouldn't be serialized remember to use a name starting with underscore.

Naming Scripts

If you want to add a name to your script (which could be useful from the editor and to access it remotely), you can add the next line at the beginning of your script:

```
//@my_script_name
```

Accessing other scripts from scripts

If you want to access data from the context of another script in the scene (or call a method), first you must retrieve that script context.

To do so the best way is to get the node from the scene, get the script component from that node, and get the context from that component.

```
var node = scene.getNode("nodename");
var component = node.getComponent( LS.Components.Script ); //or ScriptFromFile, depending which compo
var foo = component.context.foo; //read context property
```

Although you are free to register the components in some global container when the context is created so they are easier to retrieve.

```
//from inside the script you want to make accesible to other scripts
LS.Globals.my_special_script_context = this;
```

Script considerations

When coding scripts for LiteScene there are several things you must take into account:

- Remember to unbind every event you bind, otherwise the editor could have erratic behaviour.
- When using `LS.Component.Script` keep in mind that when the context is created (when your global code is executed) if you try to access to the scene tree to retrieve information (like nodes) it is possible that this info is not yet available because it hasnt been parsed yet.
- Scripts only exists if they are attached to a node, when they are detached all the data not serialized will be lost.

Global Scripts

Sometimes we want to create our own Components bypassing the scripts system, this is better because it will have better performance, more control and it will be easier to use by WebGLStudio.

But this scripts must be loaded **before** the scene is loaded. The problem with regular scripts is that they are parsed during the scene construction, this could lead to a ordering problem where a node is created using a component that is not yet defined.

To solve this problem and many others, scenes can include scripts that will be loaded before the scene tree is parsed. This are called global scripts, they are executed in the global context (window) like if they were external scripts, but the files are fetched using the `LS.ResourcesManager` (so paths are relative to the resources manager root path unless they are absolute).

The list of all the global scripts is stored in `scene.global_scripts` array.

External Scripts

When we just want to load some external library to use in our code we can add it by appending the url to the external scripts array of the scene.

The list of all the external scripts is stored in `scene.external_scripts` array.

Editing LiteScene base code

The last option is to add new components to LiteScene by manually creating new component files and adding them to litescene.js.

To do that I will recommend to insert the components inside the components folder of the code structure.

You must add also the path to the component in the deploy_files.txt inside the utils, and run the script pack.sh (to create litescene.js) or build.sh (to create litescene.js and litescene.min.js).

Missing Components

If you have created your own component class from within an script and by any reason when loading a scene the system cannot find the component specified in the JSON of the scene (maybe the component changed its name, or the script wasnt loaded), the data wont be lost an it will be stored aside so it stays in the JSON if you serialize that again.

Useful API methods

To know better some useful system methods, check this guide about API methods in LiteScene

Documentation

To know more about the APIs accessible from LiteScene check the documation websites for LiteGL, LiteScene and LiteGraph.

Example of Script

```
//@InputController
//defined: component, node, scene, globals

this.moving_speed = 100;
this.rotating_speed = 90;
this.reverse_front = true;

this.onStart = function()
{
}

this.onRenderGUI = function()
{
    LS.GUI.Label([10,10,100,40], "X: " + transform.x.toFixed(2) + " Z: " + transform.z.toFixed(2));
}

this.onUpdate = function(dt)
{
  var x_axis = 0;
  var y_axis = 0;

  var gamepad = LS.Input.getGamepad(0);
  if(gamepad)
  {
    x_axis = LS.Input.getGamepadAxis(0,"LX");
    y_axis = LS.Input.getGamepadAxis(0,"LY");
  }

  if(LS.Input.Keyboard["UP"])
    y_axis -= 1;
  if(LS.Input.Keyboard["DOWN"])
    y_axis += 1;
  if(LS.Input.Keyboard["LEFT"])
    x_axis -= 1;
  if(LS.Input.Keyboard["RIGHT"])
    x_axis += 1;

  x_axis = Math.clamp(x_axis,-1,1);
  y_axis = Math.clamp(y_axis,-1,1);

  if(this.reverse_front)
    y_axis *= -1;

  node.transform.translate(0,0,y_axis * dt * this.moving_speed);
  node.transform.rotateY(-x_axis * dt * this.rotating_speed);

    node.scene.refresh();
}

this.onGamepadConnected = function()
{
  console.log("PAD!!!");
}
```

ShaderBlocks

The problem when using ShaderMaterials is that the shader defined by the user is static,
which means that nothing from the scene could affect the ShaderCode in this ShaderMaterial.

This is a problem because when rendering objects in a scene different elements from the scene could affect the way it is seen.
For instance lights contribute to the color, but also mesh deformers (blend shapes, skinning) or even atmospheric FX (Fog).

To tackle this problem ShaderCode allows to include ShaderBlocks.

A ShaderBlock its a snippet of code that could be toggled from different elements of the render pipeline.

Depending on if the ShaderBlock is enabled or disabled it will output a different fragment of code.
And because every shaderblock has its own unique number, they can be easily mapped using bit operations in a 64 bits number.

Creating a ShaderBlock

To create a ShaderBlock you must instatiate the ShaderBlock class and configure it:

First create the ShaderBlock and give it a name (this name will be the one referenced from the shader when including it):

```
var morphing_block = new LS.ShaderBlock("morphing");
```

The next step is to add the code snippets for the Vertex or Fragment shader.

When adding a code snippet you have to pass two snippets, one for when the ShaderBlock is enabled and one for when it is disabled. This is because from your shader you will be calling functions contained in this ShaderBlock, and you want the shader to have this functions even if the ShaderBlock is disabled.

```
morphing_block.addCode( GL.VERTEX_SHADER, MorphDeformer.morph_enabled_shader_code, MorphDeformer.morp
```

Register the ShaderBlock in the system by calling the register function:

```
morphing_block.register();
```

After doing this you can call it from your shader:

```
#pragma shaderblock "morphing"

void main() {
  //...
  applyMorphing( vertex4, v_normal ); //this function is defined inside the shader block
```

To activate the ShaderBlock from your javascript component you have to get the RenderInstance and add it there:

```
   var RI = node._instances[0];
   RI.addShaderBlock( morphing_block );
```

Chaining ShaderBlocks

You can chain several ShaderBlocks so one can include another one, just use the same syntaxis using pragmas.
Just be careful of not creating recursive loops.

```
//from my ShaderBlock "morphing"...
#pragma shaderblock "morphing_texture"
```

Conditional ShaderBlocks

Sometimes you want to have a ShaderBlock that can include optionaly another one based on if that other ShaderBlock is enabled or not.
This may seem strange but it is common when we have a ShaderBlock that can be affected by other ShaderBlocks, for instance lighting is a ShaderBlock but Shadowing is another ShaderBlock and there are different Shadowing techniques.

To solve this a ShaderBlock can include a ShaderBlock but instead of specifying the name, it can specify a dynamic name that will be read from the

ShaderBlock context:

```
#pragma shaderblock morphing_mode
```

This means that the shaderblock name will be extracted from that variable inside the context (in this case `morphing_mode` is the variable).

The variable name can be defined from the ShaderBlock and only will be assigned if that ShaderBlock is enabled.

```
var morphing_texture_block = new LS.ShaderBlock("morphing_texture");
morphing_texture_block.defineContextMacros( { "morphing_mode": "morphing_texture"} );
```

Conclusion

Check the ShaderMaterial, ShaderCode, ShaderManager, ShaderBlock and GLSLParser to understand better how it works.

Also check the MorphDeformer component as a complete use-case.

Creating Shaders

You can create your own shaders for your materials. This way the rendering is much faster (the pipeline doesnt have to guess the best shader for your material) and you have more control.

Shaders are stored in the `LS.ShaderCode` class and used by some classes, mainly `LS.ShaderMaterial` and some FX components.

Because shaders are usually defined by several parts (vertex shader, fragment shader, definition of external variables) they are written in a text file where every part is defined by the backlash character and the name of the part, like `\default.vs`, this blocks are called subfiles.

GLSL

Remember that shaders in WebGL are made using the GLSL programming language, not javascript. The shaders code will be sent to the GPU to be compiled to low level assembly code so it can be executed very fast. Also keep in mind that WebGL is based in OpenGL ES 2.0, it means some GLSL features may be missing.

If you want a reference about GLSL check this website.

Javascript

When creating a shader you may want to call some javascript functions to prepare the properties of the material containing the shader, for this purpose you can write a part in the file that contain that JS code, separated from the GLSL code of the shaders.

The main functions are:

- `createUniform( label, uniform_name, type, default_value, options )`: this will make the uniform with the `uniform_name` accessible from the editor and the code. The type must be of `LS.TYPES` keeping in mind that it has to be able to be passed to the shader.
- `createSampler( label, uniform_name, texture_options )`: this will make the uniform with the `uniform_name` accessible from the editor and the code.
- `createProperty( name, default_value, options )`: this will create a var that is not passed to the shader (used in conjuction with onPrepare).

The subfile to contain this calls should be called `\js`

`\js`

```
this.createUniform("Scale","u_tex_scale","number",1, {min:0, max:1}); //create a uniform for the sha
this.createSampler("Texture","u_texture", { magFilter: GL.LINEAR, missing: "white"} ); //create a sa
this.createProperty("Node",null, LS.TYPES.NODE ); //create a property not meant to be send to the sh
this.render_state.depth_test = false; //the flags to use when rendering
```

This function will be called once the shader is assigned to the material.

RenderState

Some properties for the rendering cannot be defined inside the GLSL code (like GPU flags) so they are defined in a class called `LS.RenderState` that contains all the common flags.

If you want to use an special rendering pass consider changing those, here is a list with the flags and their default types:

```
    this.front_face = GL.CCW;
    this.cull_face = true;

    //depth buffer
    this.depth_test = true;
    this.depth_mask = true; //write in depth buffer
    this.depth_func = GL.LESS;
    //depth range: never used

    //blend function
    this.blend = false;
    this.blendFunc0 = GL.SRC_ALPHA;
    this.blendFunc1 = GL.ONE_MINUS_SRC_ALPHA;
    //blend equation

    //color mask
    this.colorMask0 = true;
    this.colorMask1 = true;
    this.colorMask2 = true;
    this.colorMask3 = true;
```

RenderQueue

To stablish the rendering order you must use the `this.queue` property.

This property is a number associated to a render queue in the system. There are queues for GEOMETRY and TRANSPARENT by default. The bigger the number the later it will be rendered.

You can type your own value or use one of the enumerated options:

- `LS.RenderQueue.DEFAULT`: means no render queue specified, the system will try to guess it.
- `LS.RenderQueue.BACKGROUND`: for object that are in the background like skyboxes (value 5)
- `LS.RenderQueue.GEOMETRY`: for regular non-transparent geometry (value 10)
- `LS.RenderQueue.TRANSPARENT`: for semitransparent objects (blend activated) (value 15)
- `LS.RenderQueue.OVERLAY`: for render calls in the screen space. (value 20)

You can also add or substract to the queue number to reorder inside the same queue:

```
    this.queue = LS.RenderQueue.TRANSPARENT + 1;
```

One example setting the alpha and the rendering order:

```
\js
    this.render_state.blend = true;
    this.queue = LS.RenderQueue.TRANSPARENT;
```

Flags

Besides the render states and the render queue there are also some generic properties that materials could switch to control the behaviour during the rendering process. Flags are stored in `this.flags`. Here is a list of them:

- `cast_shadows`: tells if this material should be rendered in the shadowmaps.
- `receive_shadows`: tells if this material should read from the shadowmaps.
- `ignore_frustum`: must be set to `true` if you shader is applying any deformation per vertex (invalidating the precomputed bounding box of the mesh. If your mesh disappears suddenly when moving the camera, this is a signal that the frustum culling is not working so set it to true.

```
\js
    this.flags.cast_shadows = false;
```

onPrepare

Sometimes we want our material to perform some actions before rendering (like extracting information from the scene and send it to the shader).

To do that you can create a `onPrepare` function, this function will be called before rendering the scene, when all materials are being prepared.

Here is one example that passes the matrix of a camera to the material:

```
this.createSampler("Texture","u_texture");
//create a property Camera that we will use to pass some data about the scene to this shader
this.createProperty("Camera", null, LS.TYPES.COMPONENT);

this.onPrepare = function( scene )
{
  if(!this.Camera) //the property Camera has not been assigned in the material
    return;
  //read the this.Camera value (the string with the UID of the camera the user assigned to this mate
  //and try to find the component with that UID (the camera object itself)
  var camera = scene.findComponentByUId( this.Camera );
  if(!camera) //no component with that uid
    return;
  if(!this._uniforms.u_textureprojection_matrix)
    this._uniforms.u_textureprojection_matrix = mat4.create();
  //now we can use that info
  camera.getViewProjectionMatrix( this._uniforms.u_textureprojection_matrix );
}
```

Pragmas

You can use some special pragmas designed to allow the user to include external code, this is helpful to reuse GLSL code between different ShaderCodes.

pragma include

This is the most basic pragma an lets you import a GLSL file stored in a resource GLSL file. The content will be copyed directly:

```
#pragma include "guest/shaders/noise_functions.glsl"
```

You can also include a subfile:

```
#pragma include "guest/shaders/noise_functions.glsl:subfilename"
```

pragma shaderblock

This feature is still a Work In Progress but it lets different components in the system interact with the material by including some code (but only if the shader allows it).

To do this first the shader must accept to have the shaderblock supported by using the shaderblock pragma. And also call the functions associated by that shaderblock:

```
//global
#pragma shaderblock "skinning"

//inside the main...
//...
 applySkinning( vertex4, v_normal );
```

pragma snippet

You can include snippets of code that are stored in the snippets container of the `LS.ShadersManager`, this is used internally to avoid creating the same code several times or to update code from scripts.

```
#pragma snippet "lighting"
```

To create a snippet:

```
LS.ShadersManager.registerSnippet("mysnippet", "//code...");
```

Structs

If you plan to use the default lighting system from your shaders you need to use several structs to map vertex data, surface data and light data. You will have access by including the "light" shaderblock:

```
    #pragma shaderblock "light"
```

The next struct defines the info per vertex:

```
struct Input {
    vec3 vertex;
    vec3 normal;
    vec2 uv;
    vec2 uv1;
    vec4 color;

    vec3 camPos;
    vec3 viewDir;
    vec3 worldPos;
    vec3 worldNormal;
    vec4 screenPos;
};
```

And to fill it you must call:

```
    Input IN = getInput();
```

The next struct defines properties of the object surface:

```
struct SurfaceOutput {
    vec3 Albedo; //base color
    vec3 Normal; //separated in case there is a normal map
    vec3 Emission; //extra light
    vec3 Ambient; //amount of ambient light reflected
    float Specular; //specular factor
    float Gloss; //specular gloss
    float Alpha; //alpha
    float Reflectivity; //amount of reflection
    vec4 Extra; //for special purposes
};
```

And to get a prefilled version you must call:

```
    SurfaceOutput o = getSurfaceOutput();
```

When calling getSurfaceOutput it will be filled with this values:

```
SurfaceOutput getSurfaceOutput()
{
    SurfaceOutput o;
    o.Albedo = u_material_color.xyz;
    o.Alpha = u_material_color.a;
    o.Normal = normalize( v_normal );
    o.Specular = 0.5;
    o.Gloss = 10.0;
    o.Ambient = vec3(1.0);
    return o;
}
```

And the next struct defines properties of the light:

```
struct FinalLight {
    vec3 Color;
    vec3 Ambient;
    float Diffuse; //NdotL
    float Specular; //RdotL
    vec3 Emission;
    vec3 Reflection;
    float Attenuation;
    float Shadow; //1.0 means fully lit
};
```

And to apply the final lighting:

```
    FinalLight LIGHT = getLight();
    LIGHT.Ambient = u_ambient_light;
    final_color.xyz = computeLight( o, IN, LIGHT );
```

Shader Example

Here is a complete shader with normalmap and specular map that support multiple lights and shadowmaps using the built-in shaderblock system so you do not have to worry about it.

```
\js
//define exported uniforms from the shader (name, uniform, widget)
this.createUniform("Number","u_number","number");
this.createSampler("Texture","u_texture");
this.createSampler("Spec. Texture","u_specular_texture");
this.createSampler("Normal Texture","u_normal_texture");
this._light_mode = 1;

\color.vs

precision mediump float;
attribute vec3 a_vertex;
attribute vec3 a_normal;
attribute vec2 a_coord;

//varyings
varying vec3 v_pos;
varying vec3 v_normal;
varying vec2 v_uvs;

//matrices
uniform mat4 u_model;
uniform mat4 u_normal_model;
uniform mat4 u_view;
uniform mat4 u_viewprojection;

//globals
uniform float u_time;
uniform vec4 u_viewport;
uniform float u_point_size;

#pragma shaderblock "light"
#pragma shaderblock "morphing"
#pragma shaderblock "skinning"

//camera
uniform vec3 u_camera_eye;
void main() {

    vec4 vertex4 = vec4(a_vertex,1.0);
    v_normal = a_normal;
    v_uvs = a_coord;
```

```
  //deforms
  applyMorphing( vertex4, v_normal );
  applySkinning( vertex4, v_normal );

    //vertex
    v_pos = (u_model * vertex4).xyz;

  applyLight(v_pos);

    //normal
    v_normal = (u_normal_model * vec4(v_normal,0.0)).xyz;
    gl_Position = u_viewprojection * vec4(v_pos,1.0);
}

\color.fs

precision mediump float;

//varyings
varying vec3 v_pos;
varying vec3 v_normal;
varying vec2 v_uvs;

//globals
uniform vec3 u_camera_eye;
uniform vec4 u_clipping_plane;
uniform float u_time;
uniform vec3 u_background_color;
uniform vec4 u_material_color;
uniform float u_number;

uniform sampler2D u_texture;
uniform sampler2D u_specular_texture;
uniform sampler2D u_normal_texture;

#pragma shaderblock "light"

#pragma snippet "perturbNormal"

void main() {
  Input IN = getInput();
  SurfaceOutput o = getSurfaceOutput();
  vec4 surface_color = texture2D( u_texture, IN.uv ) * u_material_color;
  o.Albedo = surface_color.xyz;
  vec4 spec = texture2D( u_specular_texture, IN.uv );
    o.Specular = spec.x;
    o.Gloss = spec.y * 10.0;
    vec4 normal_pixel = texture2D( u_normal_texture, IN.uv );
  o.Normal = perturbNormal( IN.worldNormal, IN.worldPos, v_uvs, normal_pixel.xyz );

  vec4 final_color = vec4(0.0);
  Light LIGHT = getLight();
  final_color.xyz = computeLight( o, IN, LIGHT );
  final_color.a = surface_color.a;

    gl_FragColor = final_color;
}
```

Tweening

Tweening is the process by which we change a value from the current value to a target value smoothly over time using an interpolation curve.

Using tweening helps to make the changes in our application much more elegant.

LiteScene comes with its own tweening system.

To tween just call the LS.Tween.easyProperty passing the object containing the property, the property name in string format, the target value, and the time the transition should last.

```
LS.Tween.easeProperty( node.transform, "x", 2.5, 1 );
```

By default it will use EASE_IN_OUT_QUAD interpolation function (quadratic interpolation for in and out), but you can choose any of the supported functions:

```
LS.Tween.easeProperty( node.transform, "x", 2.5, 1, LS.Tween.EASE_IN_CUBIC );
```

Here is a list:

- EASE_IN_QUAD:
- EASE_OUT_QUAD:
- EASE_IN_OUT_QUAD:
- QUAD:
- EASE_IN_CUBIC:
- EASE_OUT_CUBIC:
- EASE_IN_OUT_CUBIC:
- CUBIC:
- EASE_IN_QUART:
- EASE_OUT_QUART:
- EASE_IN_OUT_QUART:
- QUART:
- EASE_IN_SINE:
- EASE_OUT_SINE:
- EASE_IN_OUT_SINE:
- SINE:
- EASE_IN_EXPO:
- EASE_OUT_EXPO:
- EASE_IN_OUT_EXPO:
- EXPO:
- EASE_IN_BACK:
- EASE_OUT_BACK:
- EASE_IN_OUT_BACK:
- BACK:

In case you want to call a callback once per update or once it finishes you can pass the callbacks in order:

```
LS.Tween.easeProperty( node.transform, "x", target, 1, LS.Tween.EASE_IN_CUBIC, on_complete, on_p
```

Update

The pending tweens are processed using the method LS.Tween.update which is called automatically from LS.Player.update.

Also the system will check for a `mustUpdate` variable in the object and if it is pressed it will set it to true.

Keep in mind that if the scene is not running the tweens wont be processed.

Useful API methods

When coding scripts in LiteScene we need to access many features present in the LiteScene system (to retrieve an scene node, a component, a resource, etc.

To do so here is a guide of the most common methods that you need to know.

Remember that if you need more specific info you can check the API reference in detail in [the webglstudio documentation page]
for all the libraries involved in the WebGLStudio ecosystem.

LS.GlobalScene

To access the current active scene there is a global variable called `LS.GlobalScene`.
Feel free to access it whenever you need to retrieve anything from the scene.

```
var time = LS.GlobalScene.time;
```

All the nodes in the scene are children of one root node, in case you want to crawl the scene manually.

```
LS.GlobalScene.root
```

Retrieving nodes

To get the parent of a node you just need to use the `node.parentNode` property.

And if you want the child nodes then you can use the array `node.childNodes` or if you want all the descendants you can call `node.getDescendants` (this will retrieve children and children of children...)

When we want to retrieve one node from the scene we have different options:

- `LS.GlobalScene.getNode( name_or_uid )`: this allows to pass any kind of node identifier (like node name or uid), it is slower when working with node names because they are not indexed.
- `LS.GlobalScene.getNodeByUid( uid )`: this search by uid and because uids are indexed this is the fastest one.
- `LS.GlobalScene.getNodeByName( name )`: this search by name, but because they are not indexed this is slower.

Sometimes we want to find a node searching only inside one node. When searching using findNode the method crawls the tree so it could be slow if there are many child nodes.

- `node.findNode( name_or_uid )`
- `node.findNodeByName( name )`
- `node.findNodeByUid( name )`

Retrieving components

When we have a node sometimes we need to fetch for an specific component of that node.

All the components of a node are stored in an array at `node.components` but if we want to fetch it:

- `node.getComponent( "MeshRenderer" )` we can fetch by component class name (it will return the first occurrence).
- `node.getComponent( LS.Components.MeshRenderer )` we can fetch by class (this is slightly faster).
- `node.getComponentByUId( uid )` we can fetch by the uid of the component class.

If we have the UID of a component that we know it is in the scene we can try to find it using:

- `LS.GlobalScene.findComponent( uid )`

Retrieving Resources

Usually we need to retrieve resources like textures, meshes or data to use it in our code.

The problem with resources is that they may or may not have been loaded in the system so you need to ask the system to load them:

```
var resource = LS.ResourcesManager.load( resource_path, on_loaded );
if( resource )
{
  //...
}

function on_loaded( resource )
{
  //...
}
```

Or if you just want to retrieve it:

```
var res = LS.ResourcesManager.getResource( url );
```

Destroying instances

Sometimes you want to remove some element from the system:

- For nodes: `node.destroy();`
- For components: `node.removeComponent( component );`
- For resources: `LS.ResourcesManager.unregisterResource( filename );`

WBin

WBin is a file format to store binary data, it is simple to use, to parse and supports any binary data (from meshes, to textures, or animation).

It is similar to the WAD format, it stores a table that says the name of every chunk of data and the position and lenth inside the data block.
It also contains info about the type of data stored in that chunk (String, JSON Object, Float32Array, ...) so it can be retrieved back to the original container.

It can be used without LiteScene as an standalone binary coder/decoder.

Why another format?

LiteScene can handle many file formats, it has parsers for TGA, DAE, DDS, OBJ and STL built in.
But parsing files is a waste of time when you want to launch your app as fast as possible.

To avoid having to parse files LS allows to store data in a ready-to-use binary format, this means that all the vertex data for the meshes (which is the one that consumes most of space) is stored in plain float buffers.

This makes files very fast to parse and smaller than regular ASCII file formats but bigger than some format that include some layer of compression (like bounding box normalized 16bits floats).

The problem with local decompression is that it can take much time to perform in JS.
Using WBIN the files wont be compressed by us but they can be compressed from the server side using standard HTTP GZIP compression if you set your server to support compression for WBIN files.

Also not having compression leaves the file format very simple.

How it works?

You pass the WBin class with a JS object that contains properties, and every property will be stored inside the WBin.
If it is a typed array then it is stored as a byte block, if it is an object it is stored as a JSON String,

File format

The file has a header, then a series of lumps (every lump has a block of data associated, lumps could be strings, arrays, objects, or even other wbins).

Header: (64 bytes total)

- **FOURCC**: 4 bytes with "WBIN"
- **Version**: 4 bytes for Float32, represents WBin version used to store
- **Flags**: 2 bytes to store flags (first byte reserved, second is free to use)
- **Num. lumps**: 2 bytes number with the total amount of lumps in this wbin (max 65536)
- **ClassName**: 32 bytes to store a classname, used to know info about the object stored
- extra space for future improvements

Lump header: (64 bytes total)

- **start**: 4 bytes (Uint32), where the lump start in the binary area
- **length**: 4 bytes (Uint32), size of the lump
- **code**: 2 bytes to represent data type using code table (Uint8Array, Float32Array, ...)
- **name**: 54 bytes name for the lump

Lump binary: one single block with all the binary data...

The lump code gives info to restore the data back to its original form, here is the table of types:

- **ArrayBuffer**: "AB"
- **Int8Array**: "I1"
- **Uint8Array**: "i1"
- **Int16Array**: "I2"
- **Uint16Array**: "i2"
- **Int32Array**: "I4"
- **Uint32Array**: "i4"
- **Float32Array**: "F4"
- **Float64Array**: "F8"
- **Object**: "OB"
- **WideObject**: "WO" Object that contains string with wide chars
- **String**: "ST"
- **WideString**: "WS" String with wide chars

- **Number**: "NU" Numbers are encoded using a 64 float
- **null**: "00"

Usage

To create a WBin file from a JSON object:

```
//wbin_data is not a class, is a Uint8Array
var wbin_data = WBin.create(my_object);

//restore the data
var object = WBin.load( wbin_data );
```

You can also pass an object of an specific class if that class has a method called `toBinary` and `fromBinary`

```
var my_object = new MyClass();
var wbin_data = WBin.create( my_object ); //this will invoke my_object.toBinary()

//my new object will instance MyClass and call the method fromBinary passing the object with every da
var my_new_object = WBin.load( wbin_data );
```