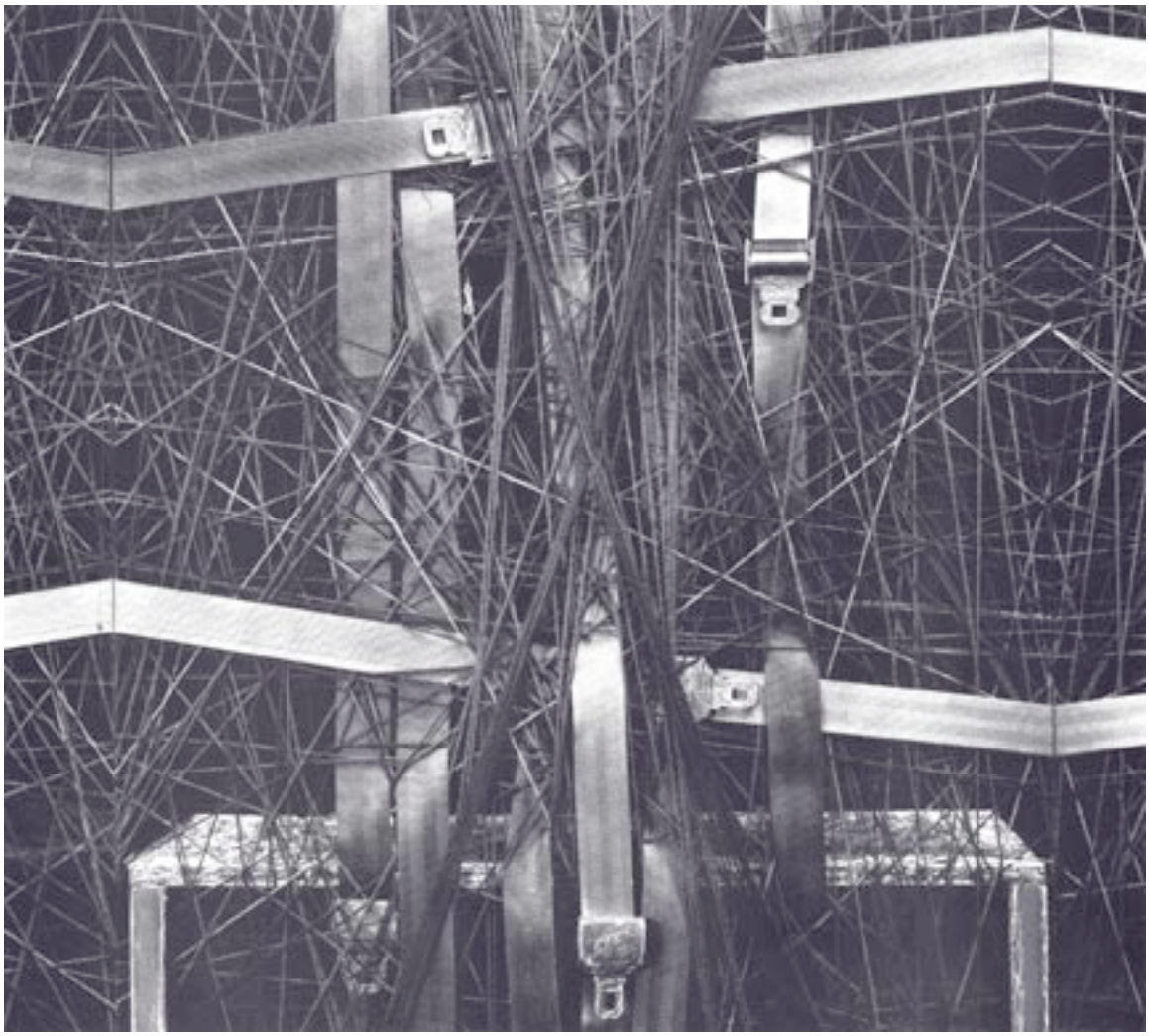
MAX



Javascript in Max

Version 4.6/7 August 2006

Table of Contents

Copyright and Trademark Notices 4
Credits 4

Basic Javascript programming for the **js** and **jsui** objects 5
How Javascript Works in the js Object 5
Overview of **js** Object Extensions to Javascript 6
Assigning a File to js and jsui 8
Arguments 8
How Input to the js Object is Handled 9
Special Function Names 9
Reserved Words 12
Global Code 12
Private (Local) Functions 13
Universally Available Methods 13
jsthis Properties 14
jsthis Methods 16
Max Properties 20
Max Modifier Key Properties 21
Max Methods 22
Patcher Constructor 22
Patcher Properties 23
Patcher Methods 24
Maxobj Properties 27
Maxobj Methods 29
Wind Properties 30
Wind Methods 31
Global Constructor 32
Global Properties 32
Global Methods 32
Accessing the Global Object from Max 32
Task Constructor 33
Task Properties 34
Task Methods 35
Folder Constructor 37
Folder Properties 37
Folder Methods 39
File Constructor 39
File Properties 39
File Methods 40
Controlling a Function's Thread of Execution 43
What's Permissible in the High-Priority Thread 45

jsui, Sketch and OpenGL 47
jsui Specific Properties 49
jsui Specific Methods 49
jsui Event Handler methods 49
Sketch Constructor 51
Sketch Properties 51
Sketch Methods 51
Shape Methods 53
Sketch Shape Attribute Methods 55
Sketch Text Methods 55
Sketch Pixel Methods 56
Sketch Stroke Methods 57
Basic 2D Stroke Style Parameters 58
Line Stroke Style Parameters 59
Sketch Setup Methods 60
Sketch OpenGL Methods 62
Image Constructor 65
Image Properties 65
Image Methods 66

Copyright and Trademark Notices

Credits

Javascript in Max Documentation: David Zicarelli and Joshua Kit Clayton

Cover Design: Lilli Wessling Hart

Graphic Design: Gregory Taylor

Basic Javascript programming for the **js** *and* **jsui** *objects*

Introduction

The **js** and **jsui** objects run Javascript version 1.5 inside of Max. This topic covers information about the functions added to the Javascript language that are specific to the **js** and **jsui** object implementation. This topic is not a general reference on Javascript programming. A good on-line resource on Javascript is available here:

http://developer.mozilla.org/en/docs/JavaScript

The Javascript language used by the **js** and **jsui** objects does not include web browser-specific functions and properties.

User interface functions and properties specific to the **jsui** object are covered in the Javascript Graphics topic. In this topic, unless specifically noted, all information covers both **js** and **jsui** even though we will refer only to the **js** object.

How Javascript Works in the js Object

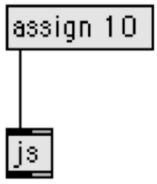
Javascript is a textual language that is "compiled" into a script. Scripts have *global code* and *functions*. Any Javascript expressions and statements that aren't inside functions are considered to be in global code.

Example:

```
var a = 2; // global code
function assign(x)
{
      a = x;      // statement in a function, not global code
}
```

Global code is executed immediately after a Javascript source file is compiled. This allows you to initialize a global variable or property. Functions can be called by global code, but they can also be called via messages sent to the **js** object. This makes implementing new Max objects in Javascript very straightforward because the name you give to a function is the same as the message name a Max user would use to invoke it.

For example, if the Javascript source above were compiled into a **js** object, we could click on the message box in the patch shown below to invoke the `assign()` function with an argument of 10.



The **js** object only uses Javascript saved in a text file. You can write Javascript code using a separate editor, or use the text window built into Max. If you use the built-in window, the **js** object will recompile its Javascript source each time you save the file. If you use another text editor, you can recompile the source from the text file by sending the message compile to the **js** object. Or you can send the message autowatch 1 to the **js** object, and every time the file is saved, the object will recompile it automatically.

Overview of js Object Extensions to Javascript

Javascript is a language designed to control the software in which it is embedded. Put another way, it is a language designed to *script* an application. Javascript code always executes within a *script context* that can add new properties and functions — this is, for example, how Netscape added browser-specific features familiar to Web developers.

In order to understand the script context in the Max **js** object, it is important to realize that there is a Javascript model of the Max world that you will be manipulating. Perhaps the most important aspect of this model is that there is a Javascript version of the Max **js** object itself. This object, which we refer to in this documentation as the *jsthis object*, is the "base" on which you build properties and methods when you write functions and global code. Some examples should make this clear. Consider the Javascript we listed above again:

```
var a = 2; // global code
function assign(x)
{
      a = x;     // statement in a function, not global code
}
```

The function `assign()` becomes a method of jsthis, and the variable `a` becomes its property. In Javascript a method can be invoked by using a dot notation on an object. We could rewrite the above example as follows using the optional Javascript `this` keyword to make the object-oriented nature of the environment more apparent. We've also added a new function `provoke()` that invokes `assign()`.

```
this.a = 2; // global code
function assign(x)
{
     this.a = x;
// statement in a function, not global code
}

function provoke()
{
     this.assign(1000);
}
```

However, you shouldn't really need to concern yourself with these semantic details if you have an irrational hatred of object-oriented terminology and concepts.

The Javascript jsthis object has certain built-in properties and methods you will need to use in order to work within the Max environment. For example, there is an `outlet()` method to send data out an outlet. You can also access the Max js object's typed-in arguments, find out which inlet received a message, and define the number of inlets and outlets the object has. These features are documented in the section on the jsthis object below.

While the word `this` in generally optional, you may need to use it when passing the current jsthis instance to functions. For example, you can send a reference to your jsthis object out an outlet, or use it in creating a Task object (see below for more about Tasks).

In addition to jsthis, the notion of a Javascript Max model is completed with a series of Javascript classes defined as part of the context. One of the primary reasons for the **js** object was to provide a more powerful interface for the patcher scripting features where you can create, connect and move new objects. The Max, Patcher, Wind, and Maxobj classes are used in scripting and Max application control. The Task object provides a convenient Javascript interface to the Max scheduler. The Folder class allows you to enumerate files. And the Global class allows different **js** instances to communicate with each other and share data, as well as interface to Max send and receive objects.

Before we enumerate the properties and functions for these Javascript classes, we'll review how you handle Javascript source text files for the **js** and **jsui** objects.

Assigning a File to js and jsui

Applies to the **js** *object only*: If you type js into an object box with no arguments, you will get an object that can't do anything and has a single inlet and outlet. The first argument to the **js** object is a filename of some Javascript source. By convention, Javascript source files end in the extension .js. In fact, if you like, you can name a file something like myscript.js and type js myscript. Assuming the file myscript.js is in the Max search path, the js object will find it.

Applies to the **jsui** *object only*: There are several ways to assign a Javascript source file to the **jsui** object.

- Select the **jsui** object. Choose **Get Info…** from the Object menu to open the **jsui** Inspector. Type the filename or use the Choose button to select it from a standard open file dialog window.

- Right-click or control-click on a selected **jsui** object to see a contextual menu. You will see several choices specific to the **jsui** object, including the names of Javascript source files in the jsui-library folder inside the Cycling '74 folder. Choose one of these files, or choose **Load New File…** to select another file using a standard open file dialog window.

- Right-click or control-click on a selected **jsui** object and click to see a contextual menu. Choose **Open Editor** from the contextual menu. After typing some Javascript, save the text window. The name you gave the file is now assigned to the **jsui** object and will be recorded when the patcher containing the **jsui** is saved.

Basic Techniques

In this section, we describe some general information about writing Javascript code for the **js** and **jsui** objects.

Arguments

After supplying a filename after the word js, you can type in additional arguments; these are available to your script's global code or any function in an array property called `jsarguments[]`. `jsarguments[0]` is the filename of your script, `jsarguments[1]` is the first typed-in argument. `jsarguments.length` is the number of typed-in arguments plus one. In other words, if there are no typed-in arguments, the value of `jsarguments.length` will be 1.

The **jsui** inspector contains an arguments field: enter the arguments you wish to use here. The first argument you type into the field can be accessed as `jsarguments[1]`.

How Input to the js Object is Handled

For most messages, a message received in an inlet of the Max **js** object will invoke a method with the same name defined for the Javascript jsthis object, passing anything after the beginning symbol as arguments to the function. Within Max, sending the message foo 1 2 3 to the **js** object invokes the `foo()` method of the jsthis object; in other words, it looks for a function property of jsthis called `foo`, passing 1, 2, and 3 as arguments to the function. If foo were defined as follows, the output in the Max window would be 1 2 3.

```
function foo(a,b,c)
{
	post(a,b,c);
}
```

`post()` is a function of the **js** object that writes the value of one or more Javascript items to the Max window, described in more detail in the jsthis Methods section below.

Special Function Names

msg_int, msg_float

To define a function to respond to a number, you need to name the function `msg_int` or `msg_float`. Example:

```
function msg_int(a)
{
	post(a);
}
```

If you define only `msg_int()`, any float received will be truncated and passed to `msg_int()`. Similarly, if only `msg_float()` exists, an int received will be passed to the `msg_float()` function.

list

To handle Max lists, i.e., a message that begins with a number, call your function list. In implementing a list function, you'll probably want to use the Javascript `arguments` property, since otherwise you couldn't handle input of varying length. Example:

```
function list(a)
{
post("the list contains",arguments.length,
"elements");
}
```

You can define an `anything()` function that will run if no specific function is found to match the message symbol received by the **js** object. If you want to know the name of the message that invoked the function, use the `messagename` property. If you want to know what inlet received the message, use the `inlet` property. Both of these properties are discussed below in the jsthis Properties section.

loadbang

To invoke a function when a patcher file containing the **js** or **jsui** object is loaded, define a function called `loadbang()`. This function will not be called when you instantiate a new **js** or **jsui** object and add it to a patcher; it will only be called when a pre-existing patcher file containing a **js** object is loaded—in other words, at the same time that **loadbang** objects in a patcher are sending out bangs. You may wish to test the loadbangdisabled property of the max object and do nothing in your loadbang function if it is true. See the section entitled The Max Object below for more information.

getvalueof

Defining a function called `getvalueof()` permits **pattr** and related objects to attach to and query an object's current value. The value of an object returned can be a Number, a String, or an Array of numbers and/or Strings. Example:

```
var myvalue = 0.25;
function getvalueof()
{
     return myvalue;
}
```

setvalueof

Defining a function called `setvalueof()` permits **pattr** and related objects to attach to and set an object's current value, passed as argument(s) to the function.

Values passed will be of type Number or String. For a value that consists of more than one Number or String, the `setvalueof()` method will receive multiple arguments. The jsthis object's `arrayfromargs()` method is useful to handle values that can contain a variable number of elements. Example:

```
function setvalueof(v)
{
     myvalue = v;
}
```

save

Defining a function called `save()` allows your script to embed state in a patcher file containing your js object. You can then restore the state when the patcher is reloaded.

Saving your state consists of storing a set of messages that your script will receive shortly after the js object containing it is recreated. These messages are stored using a special method of jsthis called embedmessage that only works inside your save function. An example will make this scheme clearer.

Suppose you have a message cowbells that sets the number of cowbells your object currently has.

```
var numcowbells = 1;

function cowbells(a)
{
     numcowbells = a;
}
```

When the patch containing the js object is saved, you would like to preserve the current number of cowbells, so you define a save() function as follows:

```
function save()
{
     embedmessage("cowbells",numcowbells);
}
```

Suppose the saved number of cowbells is 5. When it is reloaded, the js object will call your cowbell function with an argument of 5.

The first argument to embedmessage is the name of the function you want to call as a string. Additional arguments to embedmessage supply the arguments to this function. These additional arguments will typically be the values of the state you want to save.

See the description of the embedmessage message to the jsthis object below for additional information.

Reserved Words

The Max **js** object also responds to special messages that control the object itself, such as open, read, etc. Refer to the **js** and **jsui** pages in the Max Reference manual for details. If you define, for example, a function named read inside your script, it can't be executed directly via a message sent to the **js** object. However, you can still name a function `read` and call the function from another Javascript function you define.

Global Code

Global code, as we've mentioned before, consists of any Javascript statements that exist outside of any function definition. Your global code is always executed when your script is loaded or recompiled after you edit it. When a **js** or **jsui** object is being created, the global code is executed before the object is completely created. It won't have any inlets or outlets, nor does it know about its context within a patcher. This means you can use the global code to define how many inlets and/or outlets you'd like to have. However, it also means that, since outlets don't exist yet, you can't use them. If you want to perform some kind of initialization after your object has outlets, you'll need to write a `loadbang()` function mentioned in the previous section.

What to do in your global code:

- Set the number of inlets and outlets you would like (see **js** Object Properties)
- Access the arguments the user typed in with the `jsarguments[]` property
- Set up the assistance for your inlets and outlets
- Create and initialize global variables
- Use the Max object to access and control the global application environment
- Declare function properties local (see below) and immediate (discussed in the Controlling a Function's Thread of Execution section below).

What not to do:

- Send things out your outlets
- Refer to your object's Patcher (see below for the capabilities of the Patcher object)

Private (Local) Functions

If you do not want a method to be invoked outside of Javascript via a message to **js** from Max, you can set its local property to 1. For example, suppose the function `foo()` is not something we wish to expose to the outside world.

```
foo.local = 1;
function foo()
{
      post("what does Pd *really* stand for?");
}
```

Now, when we send the message foo to the **js** object, we see the following error in the Max window:

error: js: function foo is private

Universally Available Methods

The following methods are defined in the global Javascript context and can be used anywhere, including global code.

messnamed (*Max object name, message name, any arguments*)

Sends a message to the named Max object. A named Max object is an object associated with a global symbol (not an object with a patcher-specific name). For example, Max receive objects are bound to global symbols. The following code would send the message bang to the named object *flower*.

```
messnamed("flower","bang”);
```

cpost (*any arguments*)

Prints a message to the system console window. See post() below for further details about arguments.

post (*any arguments*)

Prints a representation of the arguments in the Max window. If `post()` has no arguments, it prints starting on the next line. Otherwise it prints the input on the current line separated by spaces. Arrays are unrolled to one level as with outlet.

Example:

```
a = new Array(900,1000,1100);
post(1,2,3,"violet",a);
post();
post(4,5,6);
```

These statements produce the following output in the Max window:

1 2 3 violet 900 1000 1100

4 5 6

If you want a line break in the middle of a call to `post()` you can use "\n" within a string (this is now a general feature of Max). Also, `post()` begins a new line if the last person to write to the Max window was not the **js** object.

The jsthis Object

The jsthis object is the `this` within the context of any function you define that can be invoked from Max as well as your global code. When you define functions, they become methods of your extension of jsthis. When you use variables in your global code, they become its properties. The jsthis object has certain built-in properties and methods that facilitate interacting with and controlling the Max environment.

jsthis Properties

autowatch (Number, get/set)

Turns on the automatic file recompilation feature where a file is reloaded and recompiled if it changes. This is particularly useful during development of your Javascript code if you have several **js** instances using the same source file and you want them all to update when the source changes. It can also be used to facilitate the use of an external text editor. When the text editor saves the file, the **js** object will notice and recompile it. By default, the value of `autowatch` is 0 (off). If you want to turn on autowatch, it is best to do so in your global code.

box (Maxobj, get)

Returns the Maxobj containing the **js** object. This is most useful for the **jsui** object to obtain the rectangle of the object's box. See the Maxobj object section below for more information on this object.

editfontsize (Number, get/set)

Controls the size of the font shown in the text editing window where you edit a script in points. By assigning the editfontsize property in your global code, you can override the default font size setting for text editing, which is the same size as the text shown in the Max window.

inlet (Number, get)

During the execution of a function, the `inlet` property reports the inlet number that received the message that triggered the function, starting at 0 for the leftmost inlet. This property's value is 0 within global code.

inlets (Number, get/set)

Specifies how many inlets an instance should have. The `inlets` property must be set in the global code to have any effect. If it isn't set, an object with one inlet will be created.

inspector (Number get/set)

Specific to the jsui object. The inspector property, if set to 1, causes Max to look for an inspector patch specific to your script rather than the default jsui-insp.pat file. The name used will be the name of your script (without the .js extension) plus –insp.pat. For example, if your script is called foo.js, your inspector file should be named foo-insp.pat. Inspector patches can be placed anywhere in the Max search path.

jsarguments (Array, get)

Allows access to the arguments typed into your object when it was instantiated. The filename is `jsarguments[0]`, the first typed-in argument is `jsarguments[1]`. The number of arguments plus one is `jsarguments.length`. `jsarguments[]` is available in global code and any function. It never changes after an object is instantiated, unless the Max **js** object receives the jsargs message with new typed-in arguments.

Example:

Creating an object with a variable number of outlets based on an argument typed into the **js** object:

```
// set a test default value, protects against
// a bad arg or no args
outlets = 0;
if (jsarguments.length >= 2)
      outlets = jsarguments[1];
if (!outlets)
      outlets = 1; // default value
```

max (Max, get)

Returns a Javascript representation of the "max" object (i.e., the recipient of ; max preempt 1 in a message box). Lets you send any message to the object that controls the Max application. In addition, the max object has **js** -specific properties listed in

the section on **js** Max Object Properties below. Here is an example of sending the max object the preempt message to turn Overdrive on:

```
max.preempt(1);
```

maxclass (String, get)

Returns "js" (the standard Javascript `class` property returns "jsthis")

messagename (String, get)

The name of the message to the **js** object that invoked the method currently running. In global code, this is a nil value. This is generally useful only from within an `anything()` function that will be called when no specific function name matches the message sent to the **js** object. Here is an example of an `anything()` function that adds a property to a variable declared in global code. Note the use of the tricky Javascript bracket notation to specify a variable property.

```
var stuff;
function anything(val)
{
      if (arguments.length) // were there any arguments?
            stuff[messagename] = val;
}
```

patcher (Patcher, get)

Access to the patcher containing the **js** object. See the Patcher object section below for more information on this object.

outlets (Number, get/set)

The number of inlets an object should have. The `outlets` property must be set in the global code to have any effect. If it isn't set, and object with one outlet will be created.

jsthis Methods

The important methods of the jsthis object are `outlet()` and `post()`. The others listed here are typically for more advanced applications.

arrayfromargs (*message, argument list*)

A utility for writing functions that take a variable number of arguments, and/or those that can be called using various messages (such as an anything function). The Function object has an `arguments` property that can be numerically indexed like an Array but is not an instance of Array. This means that you cannot call

Array functions such as `sort()` on the `arguments` property, or send the `arguments` property out an outlet as a list of values. The `arrayfromargs()` method will convert the `arguments` property to an Array, optionally with *message* as the zeroth element of the array. This *message* usage is useful for processing messages as though they are lists beginning with a symbol, as would be typical in your anything function. Here is an example of a function that allows its arguments to be sorted. Note that `messagename` is a property of the jsthis object that returns the name of the message that invoked the function.

```
function anything()
{
      var a = arrayfromargs(messagename,arguments);

      a.sort();
      outlet(0,a);
}
```

assist (*any arguments*)

Sets the patcher assist string for a designated inlet or outlet of a **js** object box

designed to be called from the assistance function specified as an argument to the `setinletassist()` or `setoutletassist()` method (see example under `setoutletassist()` below).

declareattribute (*attributenamex, gettername, settername, embed*)

Declare an attribute which can be set, queried, and optionally stored in the patcher file. The *attributename*, argument is required, but the following arguments are optional. If no getterr or setter methods are specified, default ones will be used. These attributes can also be referenced by pattr. A few example uses are below.

```
// default getter/setter
var foo=2;
declareattribute("foo"); //simple

// default getter/setter and embed on save
declareattribute("foo",null,null,1);

// explicit getter/setter
declareattribute("foo","getfoo","setfoo");
```

```
function setfoo(v)
{
	foo = v;
}

function getfoo()
{
	return foo;
}

function bang()
{
	outlet(0,foo);
}
```

embedmessage (method_name as string, any arguments)

The embedmessage method works only inside of your save() function. You use it to specify the name of a function you want to be called when the js object containing your script is recreated. The function name must be given as a string, followed by the arguments you wish to pass to the function. These arguments will typically be numbers, arrays, or strings (Javascript objects cannot be used in this context) and will reflect the current state of your object.

You may use the embedmessage method as many times as you want to specify multiple functions you wish to invoke to restore object state. Here is an example where functions we assume you've defined called numchairs(), numtables(), and roomname() are used in separate embedmessage statements within a save function.

```
function save()
{
	embedmessage("numchairs",20);
	embedmessage("numtables",2);
	embedmessage("roomname","diningroom");
}
```

When the js object containing this script is recreated, the function numchairs will be called with an argument of 20, followed by the numtables function with an argument of 2. Finally, the roomname function will be called with an argument of the String *diningroom*.

notifyclients()

Notifies any clients (such as the **pattr** family of objects), that the object's current value has changed. Clients can then take appropriate action such as sending a **js** instance the message getvalueof to invoke the `getvalueof()` method (if defined – see the special function names section above for more information). The `notifyclients()` method is useful for objects that define `setvalueof()` and `getvalueof()` functions for **pattr** compatibility.

outlet (*outlet_number, any arguments*)

Sends the data after the first argument out the outlet specified by the *outlet_number*. 0 refers to the leftmost outlet. If the *outlet_number* is greater than the number of outlets, no output occurs.

Example:

```
outlet(0,"bang");  // sends a bang out the left outlet
outlet(1,4,5,6); // sends a list 4 5 6 out second-from-left
```

If the argument to `outlet()` is a Javascript object, it is passed as the Max message jsobject <jsvalue> which is the address of the object. When jsobject followed by a number is sent to a **js** object, it is parsed and checked to see if the number specifies the address of a valid Javascript object. If so, the word jsobject disappears and the function sees only the Javascript object reference.

If the argument to outlet is an array, it is unrolled (to one level only) and passed as a Max message or list (depending on whether the first element of the array is a number or string).

setinletassist (*inlet_number, object*)

Associates either a number, string, or function with the numbered inlet (starting at 0 for the left inlet). If -1 is passed as the inlet number, the *object* argument is used for all inlets. In order to produce any assistance text in the patcher window the assistance function needs to call the `assist()` method described above. See example at `setoutletassist()` below. The `setinletassist()` and `setoutletassist()` functions are best called in global code but can be called at any time. You can even replace the assistance function or string dynamically.

setoutletassist (*number, object*)

Associates either a number, string, or function with the numbered outlet (starting at 0 for the left outlet). If -1 is passed as the outlet number, the *object* argument is used for all outlets. In order to produce any assistance in the patcher, the assistance function needs to call the `assist()` method described above.

Example:

```
// assistance function
function describe_it(num)
{
      assist("this is outlet number",num);
}
// global code to set it up
setoutletassist(-1,describe_it);
```

The Max Object

The Max object can be accessed as a property of a jsthis object (see jsthis Properties above). It is intended to provide control over the application environment.

Max Properties

apppath (String, get)

The pathname of the Max application

frontpatcher (Patcher, get)

The Patcher object of the frontmost patcher window, or a nil value if no patcher window is visible. You can traverse the list of open patcher windows with the `next` property of a Wind object.

isplugin (Number, get)

1 if the **js** object is within a plug-in; note that this would be 1 if the **js** object was within a plug-in loaded into the **vst~** object in Max.

isruntime (Number, get)

1 if the currently executing Max application environment does not allow editing, 0 if it does.

loadbangdisabled (Number, get)

1 if the user has disabled loadbang for the currently loading patch. If your object implements a loadbang method, it can test this property and choose to do nothing if it is true.

mainthread (Number, get)

1 if the function is currently executing in the main (low-priority) thread, 0 if the function is executing in the timer (high-priority) thread. See the section entitled Controlling a Function's Thread of Execution below for more details.

os (String, get)

The name of the platform (e.g., "windows" or "macintosh")

osversion (String, get)

The current OS version number.

time (Number, get)

The current scheduler time in milliseconds—will be a floating-point value.

version (String, get)

The version of the Max application, in the following form: "451"

Max Modifier Key Properties

cmdkeydown (Number, get)

1 if the command (Macintosh) or control (Windows) key is currently held down

ctrlkeydown (Number, get)

1 if the control key is currently held down

optionkeydown (Number, get)

1 if the option (Macintosh) or alt (Windows) key is currently held down

shiftkeydown (Number, get)

1 if the shift key is currently held down

Max Methods

The Max object can be accessed as a property of a jsthis object (see jsthis Properties above). Any message you can send to the max object using the semicolon notation in a message box can be invoked within Javascript using Javascript syntax. For example, the following in a message box:

; max preempt 1

This can be expressed in Javascript as:

```
max.preempt(1);
```

For a list of current messages that can be sent to the Max object, refer to the Messages to Max topic.

The Patcher Object

The Patcher object is a Javascript representation of a Max patcher. You can find, create, modify, and iterate through objects within a patcher, send messages to a patcher that you would use with the **thispatcher** object, etc.

There are currently three ways to get a Patcher, use the Constructor, access the patcher property of a jsthis (accessed as `this.patcher`), or use the `subpatcher()` method of a Maxobj object.

Patcher Constructor

```
var p = new Patcher(left,top,bottom,right);
```

left, top, bottom, right: global screen coordinates of the Patcher window

```
var p = new Patcher();
```

Uses 100,100,400,400 as default window coordinates

Patcher Properties

box (Maxobj, get)

If the patcher is a subpatcher, the `box` property returns the Maxobj that contains it. To traverse up to the top-level patcher:

```
prev = 0;
owner = this.patcher.box;
while (owner) {
      prev = owner;
      owner = owner.patcher.box;
}
if (prev)
      post("top patcher is",prev.name);
```

count (Number, get)

Number of objects in the patcher

filepath (String, get)

The patcher's file path on disk

firstobject (Maxobj, get)

If the patcher contains objects, this is the first one in its list. You can iterate through all objects in a patcher using the `nextobject` property of a Maxobj.

name (String, get/set)

The patcher's name (its window title, without any brackets that appear for subpatchers)

locked (Boolean, get/set)

The patcher's locked state. This property is read-only in the runtime version of Max.

maxclass (String, get)

Returns "patcher"

parentclass (String, get)

Returns the Max class name of the parent object if this is a subpatcher, or a nil value if this is a top-level patcher.

parentpatcher (Patcher, get)

If the patcher is a subpatcher, this returns the parent patcher. Otherwise it returns a nil value.

scrolloffset (Array, get/set)

X/Y coordinate array for the scroll offset of a patcher is window

scrollorigin (Array, get/set)

X/Y coordinate array for the patcher's fixed origin

wind (Wind, get)

A Javascript representation of the window associated with the patcher. See the section on the Wind object below for more information.

Patcher Methods

Any message to a patcher that you can send in Max (via the **thispatcher** object) you can send in Javascript in **js**.

Examples:

```
p = this.patcher;
p.fullscreen(1); // makes the patcher take up the whole
screen
p.dirty();       // make an editable patcher dirty
```

The Patcher methods listed below present a slighly more usable implementation of patcher scripting. You can still script a patcher using the script message, since, as shown above, a Javascript Patcher object can accept any message you can send to a **thispatcher** object.

newobject (*classname,params*)

Creates a new object of Max class *classname* in a patcher using the specified parameters and returns a Maxobj (see below) that represents it.

Example:

```
a = patcher.newobject("toggle",122,90,15,0);
```

newdefault (*left,right,classname, additional arguments*)

Creates a new object of class *classname* in a patcher using the specified parameters and return a Maxobj (see below) that represents it.

Example:

```
a = patcher.newdefault(122,90,"toggle");
```

The `newdefault()` method also accepts additional arguments for non-user interface objects that represent the created object's typed-in arguments.

Example:

```
a = patcher.newdefault(122,90,"pack", "rgb", 255, 128, 64);
```

connect (*from_object, outlet, to_object, inlet*)

Connects two objects (of type Maxobj) in a patcher. Indices for the *outlet* and *inlet* arguments start at 0 for the leftmost inlet or outlet.

Example:

```
p = this.patcher;
a = p.newobject("toggle",122,90,15,0);
b = p.newobject("toggle",122,140,15,0);
p.connect(a,0,b,0);
```

hiddenconnect (*from_object, outlet, to_object, inlet*)

Connects two objects (of type Maxobj) in a patcher with a hidden patch cord. Arguments are the same as for the connect message above.

disconnect (*from_object, outlet, to_object, inlet*)

Disconnects an existing connection between two objects (of type Maxobj) in a patcher. Indices for the *outlet* and *inlet* arguments start at 0 for the leftmost inlet or outlet.

Example (assuming the `connect()` example above):

```
p.disconnect(a,0,b,0);
```

apply (*function*)

For all objects in a patcher, calls the *function* with the each object's Maxobj as an argument. Does not recurse into subpatchers. The following example prints the name of each object's class in the Max window:

```
function printobj(a)
{
      post(a.maxclass);
      post();
      return true;
// iterfun must return true to continue
// iterating, else stops
}
this.patcher.apply(printobj);
```

applydeep (*function*)

Same as `apply()` except that `applydeep()` recurses into subpatchers (depth first).

applyif (*action_function, test_function*)

For all objects in a patcher, run the *test_function* for each object's Maxobj as an argument. If the *test_function* returns true, the *action_function* is executed with the Maxobj as an argument.

applydeepif (*action_function, test_function*)

Same as `applyif()` except that `applydeepif()` recurses into subpatchers

remove (*object*)

Removes the *object* (a Maxobj passed as an argument) from a patcher

getnamed (*name*)

Returns the first object found in a patcher with the given name. The name is a local name as specified by the **Name...** dialog in a patcher, not the name of a **send** or **receive** object. You can also set an object's name using the `varname` property of a Maxobj.

getlogical (*function*)

Calls the *function* on each object in a patcher, passing it as a Maxobj argument to the *function*. If the *function* returns true, the iteration stops and the Maxobj object is returned as the value of the `getlogical()` method. Otherwise `getlogical()` returns a nil value.

Example:

```
// search for an object with a negative left coordinate
function neg_left(a)
{
      r = a.rect;
// rect is a property that returns an array
      if (r[0] < 0)
            return 1;
      else
            return 0;
}
e = patcher.getlogical(neg_left);
if (e)
      e.rect[0] = 0;
```

bringtofront (*object*)

Moves the *object* to the front of the current layer to which it is assigned (either background or foreground). You can change the layer by setting the `background` property of a Maxobj.

sendtoback (*object*)

Moves the *object* to the back of the current layer to which it is assigned (either background or foreground). You can change the layer by setting the `background` property of a Maxobj.

The Maxobj Object

A Maxobj is a Javascript representation of a Max object in a patcher. It is returned by various methods of a Javascript Patcher object, such as `newobject()`.One important thing to keep in mind about a Maxobj is that it could eventually refer to an object that no longer exists if the underlying Max object is freed. The `valid` property can be used to test for this condition.

Maxobj Properties

rect (Array, get/set)

The location of an object in a patcher. When the object's rectangle is changed, it will move on screen if it is visible. The coordinates are stored in the following order: left, top, right, bottom.

maxclass (String, get)

The Max class (as opposed to the Javascript class, which is "Maxobj" and accessed via the standard class property) of the object.

patcher (Patcher, get)

The Patcher object that contains the Maxobj

hidden (Boolean, get/set)

Is the object set to be hidden in a locked patcher?

colorindex (Number, set/get)

If the object is set to use one of the standard 16 colors, this property is the index of the color

nextobject (Maxobj, get)

If there is another object after this one in the Patcher's list of objects, this property returns it, otherwise it returns a nil value

varname (String, get/set)

The patcher-specific name of the object, as set with the **Name...** dialog

canhilite (Boolean, get)

Whether the object can be selected for text entry (a **number box** would be an example of an object whose `canhilite` property returns true)

background (Boolean, get/set)

Whether the object is in the Patcher's background layer

ignoreclick (Boolean, get/set)

Whether the object ignores clicks

selected (Boolean, get)

Whether the object is selected in an unlocked patcher window.

js (jsthis, get)

If the Maxobj refers to an object is of Max class js, this returns the associated jsthis object

valid (Boolean, get)

Returns whether the Maxobj refers to a valid Max object

Maxobj Methods

Perhaps the most powerful thing about a Maxobj is that you can send any message to a Maxobj that you can send to a Max object in Max as if you were invoking a method on the object in Javascript. For example, if you had a number box Maxobj and you wanted to set its value to 23 without outputting the value, you could do this:

```
n.set(23);
```

Note that certain words such as `int`, `float`, and `delete` are keywords in Javascript, and you will need to use either array notation or the message method for such reserved words. For example:

```
n["int"] = 23;
//or
n.message("int", 23);
```

The following methods are common to all Maxobj objects.

message(*string, …*)

Sends the object the message specified by the *string*, followed by any additional arguments provided. This is useful for sending messages to object which dynamically dispatch messages with the "anything" message, as is the case for instances of **js**, **jsui**, **lcd**, and others.

help ()

Opens a help file describing the object, if it exists

subpatcher (*index*)

If the object contains a patcher, this function returns a (Javascript) Patcher object. The optional index is used for specifying an instance number, which only applies to **poly~** objects. If the object does not contain a subpatcher, a nil value is returned.

understands (*string*)

Returns a Boolean value if the object has an entry in its message list for the message specified by the *string*. If the entry is not a message that can be sent by a user within Max (i.e., it's a C-level "untyped" message), false is returned. This

doesn't work for messages which are dynamically dispatched with the "anything" message, as is the case for instances of **js**, **jsui**, **lcd**, and others.

The Wind Object

The Wind object is a property of a Patcher that represents its window. You cannot create a new Wind or access other types of windows such as that of a Max table object.

Wind Properties

assoc (Patcher, get)

The Patcher object associated with the window.

assocclass (String, get)

The Max class of the object associated with the window.

dirty (Boolean, get/set)

Has the window's contents been modified? This property is read-only in the runtime version of Max.

hasgrow (Boolean, get/set)

Does the window have a grow area?

hashorizscroll (Boolean, get)

Does the window have a horizontal scroll bar?

hasvertscroll (Boolean, get)

Does the window have a vertical scroll bar?

haszoom (Boolean, get/set)

Does the window have a zoom box?

hastitlebar (Boolean, get/set)

Does the window have a window title bar?

location (Array, get/set)

An array of four coordinates (left, top, right, bottom) representing the window's location in global coordinates.

next (Wind, get)

The Wind object of the next patcher visible in the application's list of windows The first Wind object can be accessed using the `frontpatcher` property of the Max object (as `max.frontpatcher.wind`).

size (Array, get/set)

An array of two coordinates (width, height) representing the window's size.

title (String, get/set)

The window's title.

visible (Boolean, get/set)

Can you see the window?

Wind Methods

bringtofront ()

Moves the window in front of all other windows

scrollto (*x, y*)

Scrolls the window so that *x* and *y* are at the top-left corner.

sendtoback ()

Moves the window behind all other windows

setlocation (*left,top,bottom,right*)

Set the global location of the window according to the coordinates passed in as arguments

The Global Object

The Global object is a fairly generic Javascript object that allows you to share data among **js** instances by adding and accessing properties. You can also access Global object properties from Max messages completely outside of **js**. Executing methods stored in Global objects from Max is not supported. However, methods are certainly among the kinds of things you can store within a Global object.

Global Constructor

```
g = new Global(name);
```

`name` represents a String that uniquely identifies the Global.

A Global is basically a reference to a Javascript object that you can't access directly. The object is connected to the Max symbol with the name you supplied as an argument (this allows it to be accessed from Max, as we'll discuss below). Every time you access a Global, it hands off the access to the secret hidden Javascript object. This means you can create any number of Global objects in your code, in any number of **js** instances, and if they all have the same name, they will all share the same data. In this way, a Global resembles a namespace.

Example:

```
g = new Global("name");
g.bob = 12;
h = new Global("name");
post(h.bob);     // will print 12
```

Global Properties

There are no fixed properties for a Global object. Instead, as described above, you assign properties to a Global object so that they can be accessed by multiple **js** object instances.

Global Methods

sendnamed (*receive_name, property_name*)

Sends the value of the named property *property_name* to the named Max **receive** object (or other Max object) bound to the specified *receive_name* symbol.

Example:

```
g = new Global("xyz");
g.ethyl = 1000;
g.sendnamed("fred","ethyl");
```

Any **receive** objects named fred will send 1000 out their outlets.

Accessing the Global Object from Max

To use Max to send a message to a named object, type a semicolon followed by the name of the receiver and the message you want to send into a message box. To set a property of a **js** Global object, send the property name followed by one or more values (multiple

values set the value of the property to an array). Assuming you have already created a Global xyz object...

This sets the value of the `george` property to 30.

```
; xyz george 30
```

This sets the value of the `frank` property to an array of three strings containing "x" "y" and "z"

```
; xyz frank x y z
```

You can also use the message sendnamed from Max to output property values to named **receive** objects. This sends the current value of the `frank` property in the **js** Global object xyz to any **receive** objects named hubert.

```
; xyz sendnamed hubert frank
```

Note a subtle distinction. When setting property values using Max, the Javascript properties are changed but no further action happens. When using `sendnamed()`, **receive** objects take action and output the property values.

The Task Object

A task is a function that can be scheduled or repeated. You can set the arguments to the function as well as the object that will be `this` when the function is called.

Task Constructor

```
var tsk = new Task(function, object, arguments);
```

The `object` argument represents the `this` during the execution of the function. Use the `this` keyword (referring to the jsthis object) to be able to use outlets and other **js** object features. The `function` argument represents the function you want to execute, and `arguments` *(an array)* represents the arguments to pass to the function. The `object` and `arguments` arguments are optional. If not present, the parent of the function object (typically jsthis) will be assumed, and there will be no arguments supplied to the function.

Example:

```
function ticker(a,b,c)
{
     post("tick");
}
args = new Array(3);
args[0] = 1;
args[1] = 2;
args[2] = 3;
t = new Task(ticker,this,args);
```

Although the overall timing accuracy of a Task function is high, the latency between the scheduled time and the actual execution time of a Task function is variable because the function runs in a low-priority thread. Therefore you should avoid using a Task function in a time-critical operation.

Task Properties

For convenience, a Task object is a property of the function executed in a Task. To access the Task from within its function, use the following standard Javascript syntax:

```
arguments.callee.task
```

We'll show you an example of this syntax for a Task that changes its interval below.

arguments (Array, get/set)

The arguments passed to the task function. `arguments[0]` is the first argument.

function (Function, get/set)

The function that is executed in the Task. You can even change this within the task function itself.

running (Boolean, get)

Whether the Task is running or not. Within a function executing within a task, this will always be 1.

interval (Number, get/set)

The time in milliseconds between repeats of the task function. The default interval is 500 ms. Here is an example of a Task with a function that causes the Task to run 10% more slowly each time the function is called, which uses the `arguments.callee.task` syntax mentioned above:

```
function taskfun()
{
      var intv = arguments.callee.task.interval;
      arguments.callee.task.interval = intv + (intv * 0.1);
}
```

object (Object, get/set)

The object that is assigned to be the `this` in the task function. Most often this will be your jsthis object, so you can, for example access the `outlet()` method. You set up your jsthis object to be the `this` by creating a task with the keyword `this` as the first argument.

Example:

If the `object` property of a task *is* a **js** object, the following three lines of code are identical from within a task function:

```
arguments.callee.task.object.outlet(1,"bang");
outlet(1,"bang");
this.outlet(1,"bang");
```

iterations (Number, get)

The number of times the task function has been called. Outside of a task function, the value of `iterations` is always 0. The value resets each time the task is started (using the `repeat()`, `execute()`, or `schedule()` methods described in the Task Methods section).

Task Methods

repeat (*number, initialdelay*)

Repeat a task function. The optional *number* argument specifies the number of repetitions. If the argument is not present or is negative, the task repeats until it is cancelled. The optional *initialdelay* argument sets the delay in milliseconds until the first iteration.

Example:

```
tsk = new Task(this,repeater_function);
tsk.interval = 1000; // every second
tsk.repeat(3);   // do it 3 times
```

Here is a repeater function that posts its iteration count to the Max window:

```
function repeater_function()
{
      post(arguments.callee.task.iterations);
}
```

In the above example, the Max window output would be:

1

2

3

execute ()

Run the task once, right now. Equivalent to calling the task function with its arguments.

schedule (*delay*)

Run the task once, with a delay. The optional *delay* argument sets the time (in milliseconds) before the task function will be executed.

cancel ()

If a task is scheduled or repeating, any future executions are cancelled. This method can be used within a task function for a self-canceling Task. The following example is a task function that will run only once, even if it is started using the `repeat()` function.

```
function once()

{
      arguments.callee.task.cancel();
}
```

The Folder Object

The Folder object is a **js** "external object" defined in the Max object called **jsfolder**. It is used to iterate through files in a folder.

Folder Constructor

```
f = new Folder(pathname);
```

`pathname` can either be the name of a folder in the search path or a complete pathname using Max path syntax.

Example:

```
f = new Folder("patches");
// would try to find the patches folder in the search path

f = new Folder("Disk:/folder1/folder2");
// uses an absolute path
```

After creating a Folder object, you'll probably want to restrict the files you see while traversing it by setting the `typelist` property:

```
f.typelist = [ "iLaF" , "maxb" , "TEXT" ];
// typical max files
```

Check the file max-fileformats.txt inside the init folder in the Cycling '74 folder for filetype codes and their associated extensions.

As a Folder object traverses through the files, you can find out information about the current file using its file properties. You can also determine whether you've looked at all properties by testing the `end` property. The following code prints the names of all files found in the folder.

```
while (!f.end) {
      post(f.filename);
      post();
      f.next();
}
```

To finish with the Folder object, you can either delete it, or send it the `close` message if you might want to reuse it.

```
f.close ();
```

Folder Properties

Two types of properties of a Folder are available: some refer to the current file within the folder, and some refer to the Folder object's state. Most of these properties are read-only.

Folder State Properties:

end (Boolean, get)

Non-zero (true) if there are no more files to examine in the folder, or if the pathname argument to the Folder object didn't find a folder.

count (Number, get)

The total number of files of the specified type(s) contained in the folder.

pathname (String, get)

The full pathname of the folder

typelist (Array of Strings, get/set)

The list of file types that will be used to find files in the folder. To search for all files (the default), set the `typelist` property to an empty array.

Current File Properties:

filename (String, get)

The name of the current file.

moddate (Array, get)

An array containing the values year, month, day, hour, minute, and second with the last modified date of the current file. These values can be used to create a Javascript Date object.

filetype (String, get)

The four-character code associated with the current file's filetype. These codes are listed in the file max-fileformats.txt, which is located at /Library/Application Support/Cycling '74 on Macintosh and C:\Program Files\Common Files\Cycling '74 on Windows. If there is no mapping for the file's extension, a nil value is returned.

extension (String, get)

The extension of the current file's name, including the period. If there are no characters after the period, a nil value is returned.

Folder Methods

reset ()

Start iterating at the beginning of the list of files. Re-opens the folder if it was previously closed with the `close()` function.

next ()

Moves to the next file.

close ()

Closes the folder. To start using it again, call the `reset()` function.

The File Object

The File object provides a means of reading and writing files from Javasccript.

File Constructor

```
f = new File(filename, access, typelist);
```

filename *can be a file in the Max search path, an absolute path, or a relative path.* Acceptable values for *access* can be "read", "write", or "readwrite". The default value for *access* is "read". Acceptable values for typelist are four character filetype codes listed in the file max-fileformats.txt, which is located at /Library/Application Support/Cycling '74 on Macintosh and C:\Program Files\Common Files\Cycling '74 on Windows. By default, *typelist* is empty. If able to, the File constructor opens the file specified by *filename*, provided it is one of the types in *typelist*.

File Properties

access (String, get/set)

File access permissions: "read", "write", or "readwrite". By default, this value is "read".

byteorder (String, get/set)

The assumed file byteorder (endianness): "big", "little", or "native". By default, this value is "native".

eof (Number, get/set)

The location of the end of file, in bytes.

filename (String, get/set)

The current filename.

filetype (String, get/set)

The four-character code associated. See c74:/init/max-fileformats.txt for possible values.

foldername (String, get)

The absolute path to parent folder.

isopen (Boolean, get)

Is file open? A useful test to determine if the File constructor was successful in finding and opening the file.

linebreak (String, get/set)

The line break convention to use when writing lines: "dos", "mac", "unix", or "native". By default, this value is "native".

position (Number, get/set)

The current file position, in bytes.

typelist (Array, get/set)

An array file type codes to filter by when opening a file. By default, this is the empty array.

File Methods

open (*filename*)

Opens the file specified by the *filename* argument. If no argument is specified, it will open the last opened file.

close ()

Closes the currently open file.

writeline (*string*)

Writes the characters contained in the *string* argument as characters to the file, starting at the current file position, and inserts a line break appropriate to the linebreak property. The file position is updated accordingly.

readline (*maximum_count*)

Reads and returns a string containing up to *maximum_count* characters or up to the first line break as read from the file, starting at the current file position. The file position is updated accordingly.

writestring (*string*)

Writes the characters contained in the *string* argument as characters to the file, starting at the current file position. Unlike writeline(), no line break is inserted. The file position is updated accordingly.

readstring (*char_count*)

Reads and returns a string containing up to *char_count* characters as read from the file, starting at the current file position. Unlike readline(), line breaks are not considered. The file position is updated accordingly.

writebytes (*byte_array*)

Writes the numbers contained in the *byte_array* argument as bytes to the file, starting at the current file position. The file position is updated accordingly.

readbytes (*byte_count*)

Reads and returns an array containing up to *byte_count* numbers, read as bytes from the file, starting at the current file position. The file position is updated accordingly.

writechars (*char_array*)

Writes the single character strings contained in the *char_array* argument as characters to the file, starting at the current file position. The file position is updated accordingly.

readchars (*char_count*)

Reads and returns an array containing the single character strings, read as characters from the file, starting at the current file position. The file position is updated accordingly.

writeint16 (*int16_array*)

Writes the numbers contained in the *int16_array* argument as signed 16-bit integers to the file, starting at the current file position. The byteorder property is taken into account when writing these values. The file position is updated accordingly.

readint16 (*int16_count*)

Reads and returns an array containing the numbers read as signed 16-bit integers from the file starting at the current file position. The byteorder property is taken into account when reading these values. The file position is updated accordingly.

writeint32 (*int32_array*)

Writes the numbers contained in the *int32_array* argument as signed 32-bit integers to the file, starting at the current file position. The byteorder property is taken into account when writing these values. The file position is updated accordingly.

readint32 (*int32_count*)

Reads and returns an array containing the numbers read as signed 32-bit integers from the file starting at the current file position. The byteorder property is taken into account when reading these values. The file position is updated accordingly.

writefloat32 (*float32_array*)

Writes the numbers contained in the *float32_array* argument as 32-bit floating point numbers to the file, starting at the current file position. The byteorder property is taken into account when writing these values. The file position is updated accordingly.

readfloat32 (*float32_count*)

Reads and returns an array containing the numbers read as 32-bit floating point numbers from the file starting at the current file position. The byteorder property is taken into account when reading these values. The file position is updated accordingly.

writefloat64 (*float64_array*)

Writes the numbers contained in the *float64_array* argument as 64-bit floating point numbers to the file, starting at the current file position. The byteorder property is taken into account when writing these values. The file position is updated accordingly.

readfloat64 (*float64_count*)

Reads and returns an array containing the numbers read as 64-bit floating point numbers from the file starting at the current file position. The byteorder property is taken into account when reading these values. The file position is updated accordingly.

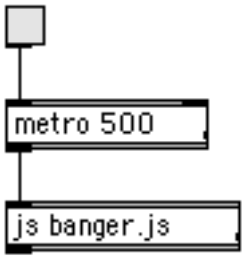
Controlling a Function's Thread of Execution

A *thread* is akin a continuously executing program running on a computer. Threads are managed by the computer's operating system: the system is constantly pausing and resuming threads to create the effect of many simultaneous activities running on a single processor. For example, you can be downloading a song from the internet in one thread while reading your e-mail in another. When the Overdrive option is turned on, Max uses two threads and asks the operating system to ensure that one of the threads, which we call the *high-priority thread*, runs as regularly as possible (usually every millisecond). In exchange, Max tries to ensure that what happens in this thread uses as little of the computer's time as possible. Time-consuming and user-interaction tasks are assigned to the *low-priority thread*.

The two threads allow you to use Max to do things that require high timing accuracy (such as MIDI) at the same time as you do things that are computationally expensive (such as decompress video) or involve user input.

Now, how does js fit into this multi-threaded scenario? By default, the js object executes all Javascript code in the low-priority thread. In particular, if it finds itself running in the high-priority thread, it will defer execution of whatever it was supposed to do to the low-priority thread. You can, however, tell js not to do this by setting the immediate property of a function.

Let's say you have a function bang that you intend to run when someone hooks up your js object to a metro, as follows:



Your bang function does something simple such as sending the value of a number out its outlet.

```
var value = 1;
function bang()
{
      outlet(0,value);
}
```

The timing accuracy of this function will be improved if you execute it immediately when the bang is received, rather than deferring it to the low-priority thread, where it will execute at some unspecified later time. In order to do this, you place the following statement in your global code.

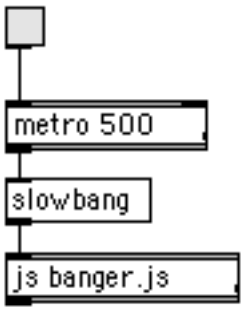
```
bang.immediate = 1;
```

However, just because a function's immediate property is set does not mean that it will always execute at high-priority. This depends on the context in which the js object received a request to execute your function. For example, if I click on a bang button connected to the js object described above, the bang function will run in the low-priority thread, because mouse clicks are always handled at low priority.

Another example that is a bit more subtle: let's say I write a function slowbang that does not have its immediate property set. It merely calls the bang function I wrote above.

```
function slowbang()
{
      bang();
}
```

Suppose we make a new patch in which the metro is hooked up to the slowbang message, as shown here:



Now the bang function will no longer execute in the high-priority thread, because slowbang was deferred to the low-priority thread before bang is executed.

You can determine the current thread of execution for your Javascript code by testing the mainthread property of the max object. The mainthread property will be 1 if the code is running in the *low-priority thread*.

In summary, a function will execute at in the high-priority thread…

• if the function's immediate property is set

and

• the js receives a message to invoke the function at high-priority

or

• the function is executing via a js Task object

What's Permissible in the High-Priority Thread

The immediate property is generally intended for functions that will perform brief computational actions that participate in the logic in your Max patch. In other words, you would use it where the js object is somewhere in the middle of the computational sequence, where preserving execution order or timing accuracy is important. The immediate property is not appropriate (and indeed, not allowed) for end-point actions such as drawing or scripting a Patcher.

The high-priority thread can be used for the following actions:

- Sending messages using the outlet function
- Posting messages to the Max window using post (although the order in which the messages appear is not guaranteed)
- Calling other Javascript functions you defined
- Accessing or controlling a js Task object
- Performing mathematical, string, or logical computation
- Accessing standard properties of the jsthis or max objects

The following actions are not guaranteed to work if you attempt to perform them in the high-priority thread. In most cases, there is no protection against doing any of these things, which may result in unexpected behavior, including Max crashing (although if you find such cases, we will do our best to prevent them).

- Any use of jsui features such as creating a Sketch or Image object or invoking methods of those objects
- Accessing files or folders using the File or Folder class
- Using the scripting or application control features in the Patcher, Maxobj, or Wind objects and their methods.

Javascript files loaded by the **jsui** object have access to the Sketch object, which may be used for drawing graphics. We will refer to the Sketch object and the methods it exposes as the **Sketch API**. The Sketch API is built upon the cross platform **OpenGL API**, and can be divided into two categories: "OpenGL Methods" and "High Level Methods". The "OpenGL Methods" are a direct binding for a large portion of the low level OpenGL API, and the "High Level Methods" are extensions built upon lower level OpenGL calls.

An example of a high level method would be `sketch.sphere(0.1)` which calculates all of the geometry and associated information for lighting and color, and based on the current state of the sketch object, issues many OpenGL calls to render a sphere. The `sketch.sphere()` method is much simpler to use than the underlying OpenGL calls it makes use of. We consider "high level" to be that which isolates the programmer from the intricate details of OpenGL.

OpenGL Conventions and Differences

All OpenGL methods begin with prefix, "gl". While we will provide a listing of supported OpenGL methods in this document, for more in-depth coverage of OpenGL we recommend that you consult either online or printed documentation concerning the OpenGL API. The www.opengl.org website is the official online resource for OpenGL, and is a good starting point for online documentation, tutorials, links, news and other information pertaining to OpenGL. There are a few important differences between the OpenGL API, and the methods which the Sketch object exposes:

1. The Sketch methods are all lowercase, and only exist within the context of a sketch object. For example, this means that glBegin() will be `sketch.glbegin()`, and glClearColor() will be `sketch.glclearcolor()`. Javascript's "with" statement may be used to avoid having to type "sketch." for every method call.

2. All symbolic constants are lowercase Javascript strings, and have no "GL_" prefix. For example the constant GL_LIGHTING will be the Javascript string "lighting", and GL_LINE_STRIP is replaced with "line_strip".

3. There are no special versions of vector calls. Only floating point values are supported, and sketch will fill in extra values with defaults. For example glColorv4fv(), glColorv3fv(), etc. will simply be `sketch.glcolor()`.

4. Sketch supports passing Javascript arrays in place of individual arguments. So glColor3f(0.,0.,1.) can be replaced with either `sketch.glcolor(0.,0.,1.)`, or `sketch.glcolor(frgb)`, where frgb is the array [0.,0.,1.].

Colors and Coordinates

As is the convention in OpenGL, color values should be specified with each component as a floating point number in the range of 0.-1., as opposed to an integer in the range 0-255. For example red would be (1.,0.,0.), rather than (255,0,0). OpenGL also supports the use of an alpha channel for transparency and other types of blending modes. Colors with alpha channel values may be specified as RGBA, for example, green with 25% opacity would be (0.,1.,0.,0.25). If there is no alpha channel value present, it is assumed to be 1.--i.e. 100% opaque. By default, alpha blending is enabled. To turn off blending, use `sketch.gldisable("blend")`. When working in 3D, depth buffering is turned on by default, and will typically interfere with attempts to blend transparent objects. To turn off depth buffering, use `sketch.gldisable("depth_test")`.

Unlike some graphics APIs, the OpenGL API does not distinguish between 2D and 3D drawing. Conventional 2D drawing is simply a subset of 3D drawing calls with specific graphics state--e.g. no lighting, no depth testing, orthorgraphic projection, et cetera. High level utility methods are provided as a convenience to setup up the OpenGL graphics state to something typically used for 2D or 3D graphics. If assuming 2D drawing conventions, one can ordinarily use z coordinates of zero for all methods that require them.

Coordinates in OpenGL are also given in terms of floating point relative world coordinates, rather than absolute pixel coordinates. The scale of these world coordinates will change depending on the current graphics transformation--i.e. translation, rotation, scaling, projection mode, viewport, etc. However, our default mapping is that Y coordinates are in the range -1. to 1 from bottom to top, and X coordinates are in the range -aspect to aspect from left to right, where aspect is equal to the ratio of width/height. In the default case, (0,0) will be center of your object, (-aspect,1.) will be the upper left corner, and (aspect,-1.) will be the lower right corner.

Note that user events are typically provided in terms of absolute screen coordinates. Please see the `sketch.screentoworld()` and `sketch.worldtoscreen()` methods for converting between absolute screen coordinates and relative world coordinates.

The jsui Object

The following section describes properties and methods that are specific to **jsui**. See the **js** Object section for properties and methods that are common to both the **js** and **jsui** object.

jsui Specific Properties

sketch (Sketch, get)

An instance of Sketch which may be drawn into. A simple example is below. See the Sketch reference section for a complete description of the properites and methods of the Sketch object.

```
function bang()
{
	sketch.glclear();
	sketch.glcolor(0.5,0.7,0.3);
	sketch.moveto(0.25,-0.25);
	sketch.circle(0.3);
	refresh();
}
```

jsui Specific Methods

refresh ()

copies the contents of `this.sketch` to the screen.

jsui Event Handler methods

Since the **jsui** object is a user interface object, it can receive and process user interface events. Currently the only user interface events which are supported are related to mouse activity and resizing off the **jsui** object. If the following methods are defined by your Javascript code, they will be called to handle these user interface events. All mouse events handlers should be defined with have a standard form of

```
function on<eventname> (x, y, button, modifier1, shift,
capslock, option, modifier2)
{
	// do something
}
```

The *modifier1* argument is the command key state on Macintosh, and the control key state on PC, and the *modifier2* argument is the control key state on Macintosh, and the right button state on PC. Modifier state is 1 if down/held, or 0 if not. If your event handler is not concerned with any trailing arguments, they can be omitted.

One potentially confusing thing is that mouse events are in absolute screen coordinates, with (0,0) as left top, and (width, height) as right bottom corners of the **jsui** object, while Sketch's drawing coordinates are in relative world coordinates, with (0,0) as the center, +1

top, -1 bottom, and x coordinates using a uniform scale based on the y coordinates. To convert between screen and world coordinates, use `sketch.screentoworld(x,y)` and `sketch.worldtoscreen(x,y,z)`. For example,

```
function onclick (x, y)
{
	sketch.glmoveto(sketch.screentoworld(x,y));
	sketch.framecircle(0.1);
	refresh();
}
```

onclick (*x, y, button, mod1, shift, caps, opt, mod2*)

If defined, will receive all initial click events. The button argument will always be on.

ondblclick (*x, y, button, mod1, shift, caps, opt, mod2*)

If defined, will receive all double click events. The button argument will always be on.

ondrag (*x, y, button, mod1, shift, caps, opt, mod2*)

If defined, will receive all dragging events. The button argument will be on while dragging, and off when the dragging has stopped.

onidle (*x, y, button, mod1, shift, caps, opt, mod2)*

If defined, will receive all idle mouse events while the mouse is over the rectangle occupied by **jsui** object. The button argument will always be off.

onidleout (*x, y, button, mod1, shift, caps, opt, mod2*)

If defined, will receive the first idle mouse event as the mouse leaves the rectangle occupied by the **jsui** object. The button argument will always be off.

onresize (*width, height*)

If defined, will receive any resize events with the new width and height.

The Sketch Object

Every instance of **jsui** has an instance of Sketch bound to the variable "sketch". This is often the only instance of Sketch you will need to use. However, if you want to do things like render sprites, have multiple layers of images, or use drawing commands to create alpha channels for images, then you can create additional instances to render in. By default, when any function in your **jsui** object has been called the context is already set for the instance of Sketch bound to the variable "sketch".

Sketch Constructor

```
var mysketch = new Sketch(); // create a new instance of
Sketch with default width and height
var mysketch = new Sketch(width,height); // create a new
instance of sketch with specified width and height
```

Sketch Properties

size (Array[2], get/set)

size[0] is width size[1] is height.

fsaa (Boolean, get/set)

Turns on/off full scene antialiasing.

Sketch Methods

Sketch Simple Line and Polygon Methods

move (*delta_x, delta_y, delta_z*)

Moves the drawing position to the location specified by the sum of the current drawing position and the delta *x*, *y*, and *z* arguments.

moveto (*x, y, z*)

Moves the drawing position to the location specified by the *x*, *y*, and *z* arguments.

point (*x, y, z*)

Draws a point at the location specified by the *x*, *y*, and *z* arguments. After this method has been called, the drawing position is updated to the location specified by the *x*, *y*, and *z* arguments.

line (*delta_x, delta_y, delta_z*)

Draws a line from the current drawing position to the location specified the sum of the current drawing position and the *delta x*, *y*, and *z* arguments. After this method has been called, the drawing position is updated to the location specified by the sum of the current drawing position and the *delta x*, *y*, and *z* arguments.

lineto (*x, y, z*)

Draws a line from the current drawing position to the location specified by the *x*, *y*, and *z* arguments. After this method has been called, the drawing position is updated to the location specified by the *x*, *y*, and *z* arguments.

linesegment (*x1, y1, z1, x2, y2, z2*)

Draws a line from the location specified by the x1, y1, and z1 arguments to the location specified by the x2, y2, and z2 arguments. After this method has been called, the drawing position is updated to the location specified by the x2, y2, and z2 arguments.

tri (*x1, y1, z1, x2, y2, z2, x3, y3, z3*)

Draws a filled triangle with three corners specified by the *x1, y1, z1, x2, y2, z2, x3, y3*, and *z3* arguments. After this method has been called, the drawing position is updated to the location specified by the *x3*, *y3*, and *z3* arguments.

frametri (*x1, y1, z1, x2, y2, z2, x3, y3, z3*)

Draws a framed triangle with three corners specified by the *x1, y1, z1, x2, y2, z2, x3, y3*, and *z3* arguments. After this method has been called, the drawing position is updated to the location specified by the *x3*, *y3*, and *z3* arguments.

quad (*x1, y1, z1, x2, y2, z2, x3, y3, z3, x4, y4, z4*)

Draws a filled quadrilateral with four corners specified by the *x1, y1, z1, x2, y2, z2, x3, y3, z3, x4, y4*, and *z4* arguments. After this method has been called, the drawing position is updated to the location specified by the *x4*, *y4*, and *z4* arguments.

framequad (*x1, y1, z1, x2, y2, z2, x3, y3, z3, x4, y4, z4*)

Draws a framed quadrilateral with four corners specified by the *x1, y1, z1, x2, y2, z2, x3, y3, z3, x4, y4*, and *z4* arguments. After this method has been called, the drawing position is updated to the location specified by the *x4*, *y4*, and *z4* arguments.

Shape Methods

circle (*radius, theta_start, theta_end*)

Draws a filled circle with radius specified by the *radius* argument. If *theta_start* and *theta_end* are specified, then an arc will be drawn instead of a full circle. The *theta_start* and *theta_en*d arguments are in terms of degrees(0-360). The current **shapeorient**, **shapeslice**, and **shapeprim** values will also affect the drawing.

cube (*scale_x*, *scale_y*, *scale_z*)

Draws a cube with width 2**scale_x*, height 2**scale_y*, depth 2**scale_z*, and center point at the current drawing position. If the *scale_y* and *scale_z* arguments are not specified, they will assume the same value as *scale_x*. The current **shapeorient**, **shapeslice**, and **shapeprim** values will also affect the drawing.

cylinder (*radius1, radius2, mag, theta_start, theta_end*)

Draws a cylinder with top radius specified by the *radius1* argument, bottom radius specified by the *radius2* argument, length specified by the *mag* argument, and center point at the current drawing position. If the *theta_start* and *theta_end* arguments are specified, then a patch will be drawn instead of a full cylinder. The *theta_start* and *theta_end* arguments are in terms of degrees(0-360). The current **shapeorient**, **shapeslice**, and **shapeprim** values will also affect the drawing.

ellipse (*radius1, radius2, theta_start, theta_end*)

Draws a filled ellipse with radii specified by the *radius1* and *radius2* arguments. If *theta_start* and *theta_end* are specified, then an arc will be drawn instead of a full ellipse. The *theta_start* and *theta_end* arguments are in terms of degrees(0-360). The current **shapeorient**, **shapeslice**, and **shapeprim** values will also affect the drawing.

framecircle (*radius, theta_start, theta_end*)

Draws a framed circle with radius specified by the *radius* argument. If *theta_start* and *theta_end* are specified, then an arc will be drawn instead of a full circle. The *theta_start* and *theta_end* arguments are in terms of degrees(0-360). The current **shapeorient**, **shapeslice**, and **shapeprim** values will also affect the drawing.

frameellipse (*radius1, radius2, theta_start, theta_end*)

Draws a framed ellipse with radii specified by the *radius1* and *radius2* arguments. If *theta_start* and *theta_end* are specified, then an arc will be drawn instead of a full ellipse. The *theta_start* and *theta_end* arguments are in terms of degrees(0-360). The current **shapeorient**, **shapeslice**, and **shapeprim** values will also affect the drawing.

plane (*scale_x1, scale_y1, scale_x2, scale_y2*)

Draws a plane with top width 2***scale_x1*, left height 2***scale_y1*, bottom width 2***scale_x*2, right height 2***scale_y2*, and center point at the current drawing position. If *scale_y1* is not specified, it will assume the same value as *scale_x1*. If *scale_x2* and *scale_y2* are not specified, they will assume the same values as *scale_x1* and *scale_y1* respectively. The current **shapeorient**, **shapeslice**, and **shapeprim** values will also affect the drawing.

roundedplane (*round_amount, scale_x, scale_y*)

Draws a rounded plane with width 2***scale_x*, and height 2***scale_y* and center point at the current drawing position. The size of the rounded portion of the plane is determined by the round_amount argument. If *scale_y* is not specified, it will assume the same value as *scale_x*. The current **shapeorient**, **shapeslice**, and **shapeprim** values will also affect the drawing.

sphere (*radius, theta1_start, theta1_end, theta2_start, theta2_end*)

Draws a sphere with radius specified by the *radius* argument and center point at the current drawing position. If the *theta1_start*, *theta1_end*, *theta2_start*, and *theta2_end* arguments are specified, then a patch will be drawn instead of a full sphere. The *theta1_start*, *theta1_end*, *theta2_start*, and *theta2_end* arguments are in terms of degrees(0-360). The current **shapeorient**, **shapeslice**, and **shapeprim** values will also affect the drawing.

torus (*radius1, radius2, theta1_start, theta1_end, theta2_start, theta2_end*)

Draws a torus with major radius specified by the *radius1* argument, minor radius specified by the *radius2* argument, and center point at the current drawing position. If *theta1_start*, *theta1_end*, *theta2_start*, and *theta2_end* are specified, then a patch will be drawn instead of a full torus. The *theta1_start*, *theta1_end*, *theta2_start*, and *theta2_end* arguments are in terms of degrees(0-360). The current **shapeorient**, **shapeslice**, and **shapeprim** values will also affect the drawing.

Sketch Shape Attribute Methods

shapeorient (*rotation_x, rotation_y, rotation_z*)

Sets the rotation for drawing internal to any of the "shape" drawing methods to the rotation specified by the *x_rot*, *y_rot*, and *rotation_x*, *rotation_y*, and *rotation_z* arguments. *Its* use internal to a given shape method such as torus(0.1) would essentially be equivalent to the following set of OpenGL calls.

```
with (sketch) {
      glmatrixmode("modelview");
      glpushmatrix();
      glrotate(rotation_x,1.,0.,0.);
      glrotate(rotation_y,1.,1.,0.);
      glrotate(rotation_z,0.,0.,1.);
      torus(0.5,0.2);
      glpopmatrix();
}
```

shapeslice (*slice_a, slice_b*)

Sets the number of slices to use when rendering any of the "shape" drawing methods. Increasing the *slice_a* and *slice_b* arguments will increase the quality at which the shape is rendered, while decreasing these values will improve performance.

shapeprim (*draw_prim*)

Sets the OpenGL drawing primitive to use within any of the "shape" drawing methods. Acceptable values for the *draw_prim* argument are the following strings: lines, line_loop, line_strip, points, polygon, quads, quad_grid, quad_strip, triangles, tri_grid, tri_fan, tri_strip.

Sketch Text Methods

font (*fontname*)

Sets the current font to the fontname specified by the *fontname* argument.

fontsize (*points*)

Sets the fontsize to the size specified by the *points* argument. Note that this size is an absolute, rather than relative value.

gettextinfo (*string*)

Returns an array containing the width and height of the given string in absolute screen coordinates, taking into account the current font and fontsize.

text (*string*)

Draws the text specified by the *string* argument at the current drawing position, taking into account the current font, fontsize, and text alignment. Text is strictly 2D, and does not take into account any world transformations. After calling the text method, if the x axis text alignment is set to "left", the current drawing position will be updated to reflect the world position associated with the end of the string. If the x axis text alignment is set to "right", the current drawing position will be updated to reflect the world position associated with the end of the string. If the x axis text alignment is set to "center", the current drawing position will remain unchanged.

textalign (*align_x, align_y*)

Sets the alignment of text to be drawn with respect to the current drawing position. Acceptable values for the x axis alignment are: "left", "right", or "center". Acceptable values for the y axis alignment are: "bottom", "top", or "center". The default alignment is "left", "bottom".

Sketch Pixel Methods

copypixels (*source_object, destination_x, destination_y, source_x, source_y, width, height*)

Copies pixels from the source object to the location specified by the *destination_x* and destin*ation_y* arguments. The initial x and y offset into the source and size of the rectangle copied can be speified by the *source_x*, *source_y*, *width* and *height* arguments. If these are not present an x and y offset of zero and width and height equal to the source image is assumed. No scaling of pixels is supported. The source object can either be an instance of Image, or Sketch. If blending is enabled in the destination sketch object, alpha blending will be performed and the current alpha color will also be applied globally. The copypixels method is much faster than obtaining the equivalent result using glbindtexture() to texture a plane, and is the recommended means of drawing images when scaling and rotation is not required.

depthatpixel (*x, y*)

Returns the depth value associated with the currently rendered pixel at a given absolute screen coordinate.

freepeer ()

Frees the image data from the native c peer, which is not considered by the JavaScript garbage collector, and may consume lots of memory until the garbage

collector decides to run based on JS allocated memory. Once called, the Sketch object is not available for any other use.

getpixel (*x, y*)

Returns an array containing the pixel value at the specified location. This array is ordered RGBA, i.e. array element 0 is red, 1, green, 2, blue, 3 alpha. Color values are floating point numbers in the range 0.-1.

setpixel (*x, y, red, green, blue, alpha*)

Sets the pixel value at the specified location. Color values are floating point numbers in the range 0.-1.

screentoworld (*x,y*)

Returns an array containing the *x*, *y*, and *z* world coordinates associated with a given screen pixel using the same the depth from the camera as 0,0,0. Optionally a third depth arg may be specified, which may be useful for hit detection and other applications. The depth value is typically specified in the range 0.-1. where 0 is the near clipping plane, and 1. is the far clipping plane. The worldtoscreen method can be used to determine the depth value of a given world coordinate, and the depthatpixel method can be used to determine the depth value associated with the currently rendered pixel at a given absolute screen coordinate.

worldtoscreen (*x, y, z*)

Returns an array containing the *x*, *y*, and depth screen coordinates associated with a given world coordinate. The depth value is typically specified in the range 0.-1. where 0 is the near clipping plane, and 1. is the far clipping plane.

Sketch Stroke Methods

beginstroke (*stroke_style*)

Begin definition of a stroked path of the style specified by the *stroke_style* argument. Currently supported stroke styles are "basic2d" and "line".

endstroke ()

End definition of a stroked path, and render the path.

strokeparam (*parameter_name, parameter_values, ...*)

Set the current value of the parameter specified by the *parameter_name* argument to be the value specified by *parameter_values* argument(s). Some parameters are global for the extent of a stroked path definition, while others may vary on a point by point basis.

strokepoint (*x, y, z*)

Defines an anchor point at the location specified by the *x*, *y*, and *z* arguments. Some stroke styles such as "basic2d" will ignore the z coordinate.

Basic 2D Stroke Style Parameters

alpha

May vary point to point. Value is specified as an alpha value. Useful if alpha is the only color channel which will vary throughout the path.

color

May vary point to point. Values are specified as red, green, blue, and alpha values.

order

Global. Value is specified as interpolation order. The default order is 3, or bi-cubic interpolation.

outline

Global. Value is specified as on/off (0/1). The default is 1.

outcolor

May vary point to point. Values are specified as red, green, blue, and alpha values. If no outcolor has been specified, then the current color is assumed.

scale

May vary point to point. Value is specified as an width value. This value controls how wide the stroked path is.

slices

Global. Value is specified as number of slices per curve section. The default is 20.

Line Stroke Style Parameters

alpha

May vary point to point. Value is specified as an alpha value. Useful if alpha is the only color channel which will vary throughout the path.

color

May vary point to point. Values are specified as red, green, blue, and alpha values.

order

Global. Value is specified as interpolation order. The default order is 3, or bi-cubic interpolation.

slices

Global. Value is specified as number of slices per curve section. The default is 20.

Sketch Setup Methods

default2d ()

The default2d method is a simple way to set the graphics state to default properties useful for 2D graphics. It is called everytime your object is resized if default2d() has been called more recently than default3d(). It is essentially equivalent to the following set of calls:

```
with (sketch) {
      glpolygonmode("front_and_back","fill");
      glpointsize(1.);
      gllinewidth(1.);
      gldisable("depth_test");
      gldisable("fog");
      glcolor(0.,0.,0.,1.);
      glshademodel("smooth");
      gldisable("lighting");
      gldisable("normalize");
      gldisable("texture");
      glmatrixmode("projection");
      glloadidentity();
      glortho(-aspect, aspect, -1, 1, -1,100.);
      glmatrixmode("modelview");
      glloadidentity();
      glulookat(0.,0.,2.,0.,0.,0.,0.,0.,1.);
      glclearcolor(1., 1., 1., 1.);
      glclear();
      glenable("blend");
      glblendfunc("src_alpha","one_minus_src_alpha");
}
```

default3d ()

The default3d method is a simple way to set the graphics state to default properties useful for 3D graphics. It is called everytime the **jsui** object is resized if `default3d()` has been called more recently than `default2d()`.

It is essentially equivalent to the following set of calls:

```
with (sketch) {
      glpolygonmode("front_and_back","fill");
      glpointsize(1.);
      gllinewidth(1.);
      glenable("depth_test");
      glenable("fog");
      glcolor(0.,0.,0.,1.);
      glshademodel("smooth");
      gllightmodel("two_side", "true");
      glenable("lighting");
      glenable("light0");
      glenable("normalize");
      gldisable("texture");
      glmatrixmode("projection");
      glloadidentity();
      gluperspective(default_lens_angle, aspect, 0.1, 100.);
      glmatrixmode("modelview");
      glloadidentity();
      glulookat(0.,0.,2.,0.,0.,0.,0.,0.,1.);
      glclearcolor(1., 1., 1., 1.);
      glclear();
      glenable("blend");
      glblendfunc("src_alpha","one_minus_src_alpha");
}
```

ortho3d ()

The orth3d method is a simple way to set the graphics state to default properties useful for 3D graphics, using an orthographic projection (i.e. object scale is not affected by distance from the camera). It is called every time the **jsui** object is resized if `ortho3d()` has been called more recently than `default2d()`, or `default3d()`.

It is essentially equivalent to the following set of calls:

```
with (sketch) {
      glpolygonmode("front_and_back","fill");
      glpointsize(1.);
      gllinewidth(1.);
      glenable("depth_test");
      glenable("fog");
      glcolor(0.,0.,0.,1.);
      glshademodel("smooth");
      gllightmodel("two_side", "true");
      glenable("lighting");
      glenable("light0");
      glenable("normalize");
      gldisable("texture");
      glmatrixmode("projection");
      glloadidentity();
      glortho(-aspect, aspect, -1, 1, -1,100.);
      glmatrixmode("modelview");
      glloadidentity();
      glulookat(0.,0.,2.,0.,0.,0.,0.,0.,1.);
      glclearcolor(1., 1., 1., 1.);
      glclear();
      glenable("blend");
      glblendfunc("src_alpha","one_minus_src_alpha");
}
```

Sketch OpenGL Methods

glbegin (*draw_prim*)

glbindtexture (*image_object*) Note: this method also calls **glenable**(*texture*)

glblendfunc (*src_function, dst_function*)

glclear ()

glclearcolor (*red, green, blue, alpha*)

glcleardepth (*depth*)

glclipplane (*plane, coeff1, coeff2, coeff3, coeff4*)

glcolor (*red, green, blue, alpha*)

glcolormask (*red, green, blue, alpha*)

glcolormaterial (*face, mode*)

glcullface (*face*)

gldepthmask (*on*)

gldepthrange (*near, far*)

gldisable (*capability*)

gldrawpixels (*image*)

gledgeflag (*on*)

glenable (*capability*)

glend ()

glfinish ()

glflush ()

glfog (*parameter_name, value*)

glfrustum (*left, right, bottom, top, near, far*)

glhint (*target, mode*)

gllight (*light, parameter_name, value*)

gllightmodel (*parameter_name, value*)

gllinestipple (*factor, bit-pattern*)

gllinewidth (*width*)

glloadidentity ()

glloadmatrix (*matrix_array*)

gllogicop (*opcode*)

glmaterial

glmatrixmode (*mode*)

glmultmatrix (*matrix_array*)

glnormal (*x, y, z*)

glortho (*left, right, bottom, top, near, far*)

glpointsize (*size*)

glpolygonmode (*face, mode*)

glpolygonoffset (*factor, units*)

glpopattrib ()

glpopmatrix ()

glpushattrib ()

glpushmatrix ()

glrect (*x1, y1, x2, y2*)

glrotate (*angle, x, y, z*)

glscale (*x_scale, y_scale, z_scale*)

glscissor (*x, y, width, height*)

glshademodel (*mode*)

gltexcoord (*s, t*)

gltexenv (*parameter_name, val1, val2, val3, val4*)

gltexgen (*coord, parameter_name, val1, val2, val3, val4*)

gltexparameter (*parameter_name, val1, val2, val3, val4*)

gltranslate (*delta_x, delta_y, delta_z*)

glulookat (*eye_x, eye_y, eye_z, center_x, center_y, center_z, up_x, up_y, up_z*)

gluortho2d (*left, right, bottom, t*op)

gluperspective (*fovy, aspect, near, far*)

glvertex (*x, y, z*)

glviewport (*x, y, width, height*)

The Image Object

The Image object can be used to draw images in an instance of the Sketch. It is possible to load image files from disk, create images from instances of Sketch, or generate them manually. The Image object has several methods to assist in manipulating images once generated. Note that alphablending is on by default in sketch. Certain file formats which contain alpha channels such as PICT or TIFF may have their alpha channel set all off. File formats which do not contain an alpha channel such as JPEG, by default have an alpha channel of all on. If you are having trouble seeing an image when attempting to draw in an instance of Sketch, you may want to either turn off blending with `gldisable("blend")`, or set the alpha channel to be all on with `clearchannel("alpha",1.)`.

Image Constructor

```
var myimg = new Image(); // create a new Image instance
with default width + height
var myimg = new Image(width,height); // create a new Image
instance with the specified witdth + height
var myimg = new Image(filename); // create a new Image
instance from a file from disk
var myimg = new Image(imageobject); // create a new Image
instance from another instance of Image
var myimg = new Image(sketchobject); // create a new Image
instance from an instance of Sketch
```

Image Properties

size(Array[2], get/set)

size[0] is width size[1] is height.

Image Methods

adjustchannel (*channel, scale, bias*)

Adjusts all channel values in the image channel specified by the *channel* argument, by multiplying the channel value by the value specified by the *scale* argument and then adding the value specified by the *bias* argument. The resulting channel is clipped to the range 0.-1. Acceptable values for the *channel* argument are the strings: "red", "green", "blue", or "alpha".

alphachroma (*red, green, blue, tolerance, fade, minkey, maxkey*)

Generates an alpha channel based on the chromatic distance from the specified RGB target color. If no tolerance, *fade* or *minkey* arguments are specified they are assumed to be 0. If no *maxkey* argument is specified, it is assumed to be 1.

blendchannel (*source_object, alpha, source_channel, destination_channel*)

Similar to the copychannel method, except supports a blend amount specified by the *alpha* argument. The source object can only be an instance of Image (not Sketch). If the source object is not the same size as the destination object, then rectangle composed of the minimum width and height of each, is the rectangle of values which will be blended. Acceptable values for the channel arguments are the strings: "red", "green", "blue", or "alpha".

blendpixels (*source_object, alpha, destination_x, destination_y, source_x, source_y, width, height*)

Similar to the copypixels method, except supports alpha blending, including a global alpha value specified by the *alpha* argument. This global alpha value is multiplied by the source object's alpha channel at each pixel. Instances of Sketch do not contain an alpha channel, which is assumed to be all on. The source object can either be an instance of Image, or Sketch.

clear (*red, green, blue, alpha*)

Sets all pixels in the image to be the value specified by the *red*, *green*, *blue*, and *alpha* arguments. If no arguments are specified, these values are assumed to be (0, 0, 0, 1) respectively.

clearchannel (*channel, value*)

Sets all channel values in the image channel specified by the *channel* argument to be the value specified by the *value* argument. If no value argument is specified, it is assumed to be 0. Acceptable values for the *channel* argument are the strings: "red", "green", "blue", or "alpha".

copychannel (*source_object, source_channel, destination_channel*)

Copies the channel values from the source object's channel specified by the *source_channel* argument to the destination object's channel specified by the *destination_channel* argument. The source object can only be an instance of Image (not Sketch). If the source object is not the same size as the destination object, then rectangle composed of the minimum width and height of each, is the rectangle of values which will be copied. Acceptable values for the channel arguments are the strings: "red", "green", "blue", or "alpha".

copypixels (*source_object, destination_x, destination_y, source_x, source_y, width, height*)

Copies pixels from the source object to the location specified by the *destination_x* and *destination_y* arguments. The initial *x* and *y* offset into the source and size of the rectangle copied can be speified by the *source_x*, *source_y*, *width* and *height* arguments. If these are not present an x and y offset of zero and width and height equal to the source image is assumed. No scaling of pixels is supported. The source object can either be an instance of Image, or Sketch.

flip (*horizontal_flip, vertical_flip*)

Flips the image horizontally and or vertically. Arguments can be 0 or 1, where 0 is no flip, and 1 is flip.

freepeer ()

Frees the image data from the native c peer, which is not considered by the JavaScript garbage collector, and may consume lots of memory until the garbage collector decides to run based on JS allocated memory. Once called, the Image object is not available for any other use.

fromnamedmatrix (*matrixname*)

Copies the pixels from the jit.matrix specified by *matrixname* to the image.

getpixel (*x, y*)

Returns an array containing the pixel value at the specified location. This array is ordered RGBA, i.e. array element 0 is red, 1, green, 2, blue, 3 alpha. Color values are floating point numbers in the range 0.-1.

setpixel (*x, y, red, green, blue, alpha*)

Sets the pixel value at the specified location. Color values are floating point numbers in the range 0.-1.

swapxy ()

Swaps the axes of the image so that width becomes height and vice versa. The effective result is that the image is rotated 90 degrees counter clockwise, and then flipped vertically.

tonamedmatrix (*matrixname*)

Copy the pixels from the image to the jit.matrix specified by *matrixname*.