The Sun Labs Lively Kernel
A Technical Overview

February 1, 2008

Summary

The *Sun Labs Lively Kernel* is a new web programming environment developed at Sun Microsystems Laboratories. The Lively Kernel supports desktop-style applications with rich graphics and direct manipulation capabilities, but without the installation or upgrade hassles that conventional desktop applications have. The system is written entirely in the JavaScript programming language – a language supported by all the web browsers – with the intent that the system can run in commercial web browsers without installation or any plug-in components. The system leverages the dynamic aspects of the JavaScript language to make it possible to create, modify and deploy applications on the fly, using tools built into the system itself. In addition to its application execution capabilities, the Lively Kernel system can also function as an integrated development environment (IDE), making the whole system self-sufficient and able to improve and extend itself dynamically.

In this document we provide a technical overview of the Sun Labs Lively Kernel, summarizing its overall architecture and core APIs. We have also included a few sample applications and widgets to demonstrate application development with the system.

1. Introduction

The widespread adoption of the World Wide Web has fundamentally changed the landscape of software development. In the past few years, the Web has become the *de facto* deployment environment for new software systems and applications. We believe that in the next 5-10 years, the vast majority of new software applications will be written for the Web, instead of any conventional target platform such as a specific operating system, CPU architecture or device. In general, the software industry is currently in the middle of a *paradigm shift* towards web-based software. In this new era of web-based software, applications live on the Web as services; they consist of data, code and other resources that can be located anywhere in the world. Unlike conventional desktop applications, they require no installation or manual upgrades.

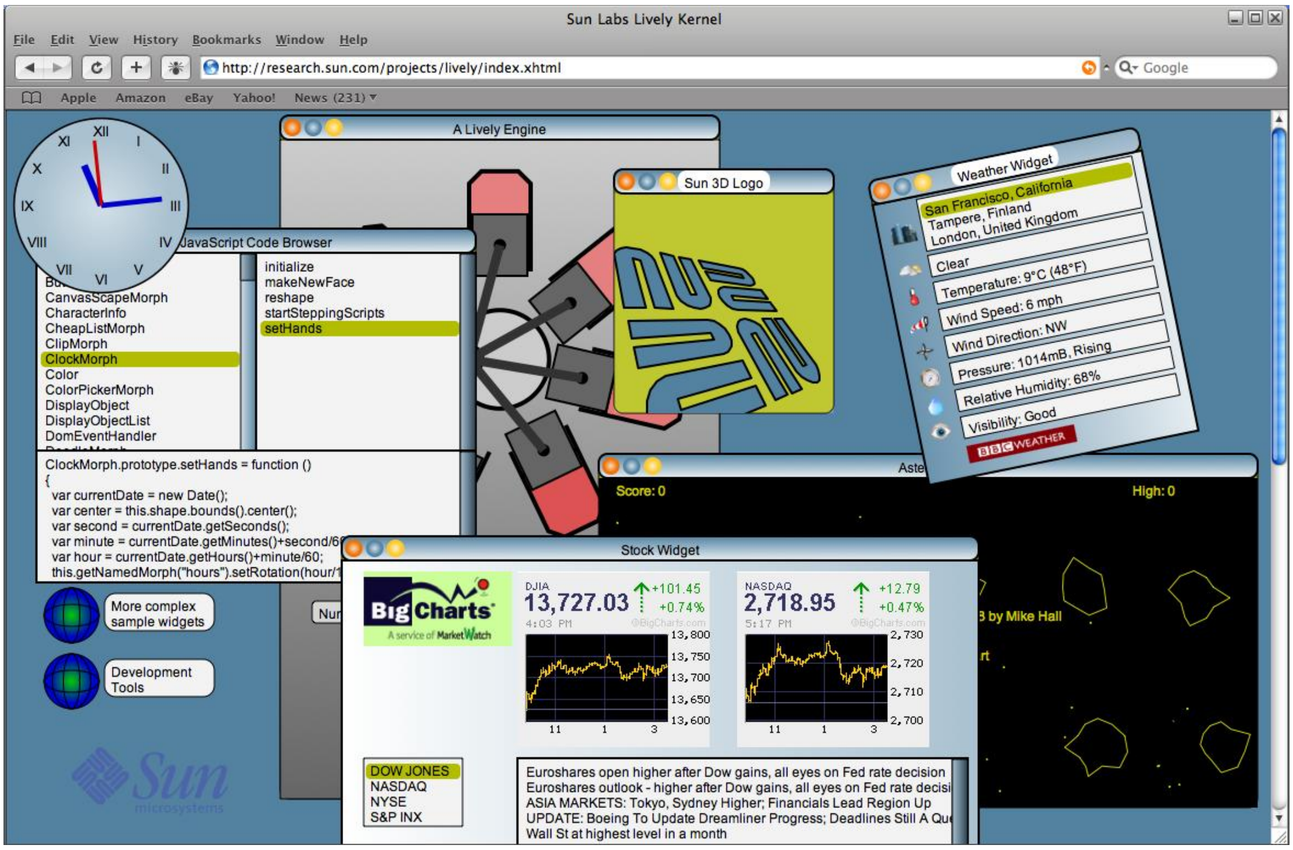
The *Sun Labs Lively Kernel* is an innovative, exceptionally interactive and dynamic system targeted specifically to web application development. The Lively Kernel supports desktop-style applications with rich graphics and direct manipulation capabilities, but without the installation or upgrade hassles that conventional desktop applications have. The system is written entirely in the *JavaScript* programming language – a language supported by all the web browsers – with the intent that the system can run in commercial web browsers without installation or any plug-in components. The Lively Kernel leverages the dynamic aspects of the JavaScript language to make it possible to create, modify and deploy applications on the fly, using tools built into the system itself. In addition to its application execution capabilities, the Lively Kernel can also function as an integrated development environment (IDE), making the whole system self-sufficient and able to improve and extend itself dynamically

In this document we provide a technical overview of the Sun Labs Lively Kernel, summarizing its overall architecture and core APIs. We have also included a few sample applications and widgets to demonstrate application development with the system.

2. Architectural Overview of the Lively Kernel

In short, the Sun Labs Lively Kernel is a fully interactive, "zero-installation" web application development environment that has been written entirely in JavaScript. It runs in an ordinary web browser without installation or plug-in components; it supports desktop-style applications with rich user interface features and direct manipulation capabilities; it enables application development and deployment in a web browser with no installation or upgrades, using nothing more than existing web technologies. In addition to its application execution capabilities, the Lively Kernel can also function as an integrated development environment (IDE).

A screen snapshot of the Lively Kernel is provided in Figure 1. In this snapshot, you can see a number of applications, widgets and tools (including a JavaScript code browser) running in a web browser simultaneously


Sun Labs Lively Kernel
File Edit View History Bookmarks Window Help
http://research.sun.com/projects/lively/index.xhtml
Google
Apple Amazon eBay Yahoo! News (231)
A Lively Engine
Sun 3D Logo
Weather Widget
JavaScript Code Browser
More complex sample widgets
Development Tools
Stock Widget
Score: 0
High: 0
by Mike Hall
DJIA 13,727.03 +101.45 +0.74%
NASDAQ 2,718.95 +12.79 +0.47%

```
CanvasScapeMorph
CharacterInfo
CheapListMorph
ClipMorph
ClockMorph
Color
ColorPickerMorph
DisplayObject
DisplayObjectList
DomEventHandler
```

```
initialize
makeNewFace
reshape
startSteppingScripts
setHands
```

```
ClockMorph.prototype.setHands = function ()
{
 var currentDate = new Date();
 var center = this.shape.bounds().center();
 var second = currentDate.getSeconds();
 var minute = currentDate.getMinutes()+second/6
 var hour = currentDate.getHours()+minute/60;
 this.getNamedMorph("hours").setRotation(hour/1
```

San Francisco, California
Tampere, Finland
London, United Kingdom

Clear
Temperature: 9°C (48°F)
Wind Speed: 6 mph
Wind Direction: NW
Pressure: 1014mB, Rising
Relative Humidity: 68%
Visibility: Good

DOW JONES
NASDAQ
NYSE
S&P INX

Euroshares open higher after Dow gains, all eyes on Fed rate decision
Euroshares outlook - higher after Dow gains, all eyes on Fed rate decisi
ASIA MARKETS: Tokyo, Sydney Higher; Financials Lead Region Up
UPDATE: Boeing To Update Dreamliner Progress; Deadlines Still A Que
Wall St at highest level in a month

Figure 1: Screen snapshot of the Lively Kernel

2.1 The Lively Kernel Architecture from the End User's Viewpoint

In the most typical situation, the user launches the Lively Kernel by typing a URL in a web browser. When the user presses the the return key to load the web page, the files that constitute the Lively Kernel system itself and the application(s) are downloaded to the web browser for execution (see Figure 2). Note that no pre-installation or plug-in components are required for the Lively Kernel itself or for the applications running in the system, apart from the JavaScript engine and the graphics libraries that already exist in the browser. Once the Lively Kernel is running, the application will communicate with the web server using asynchronous HTTP. The Lively Kernel and the application(s) will remain in the browser running until the user closes the browser or moves to another web page.

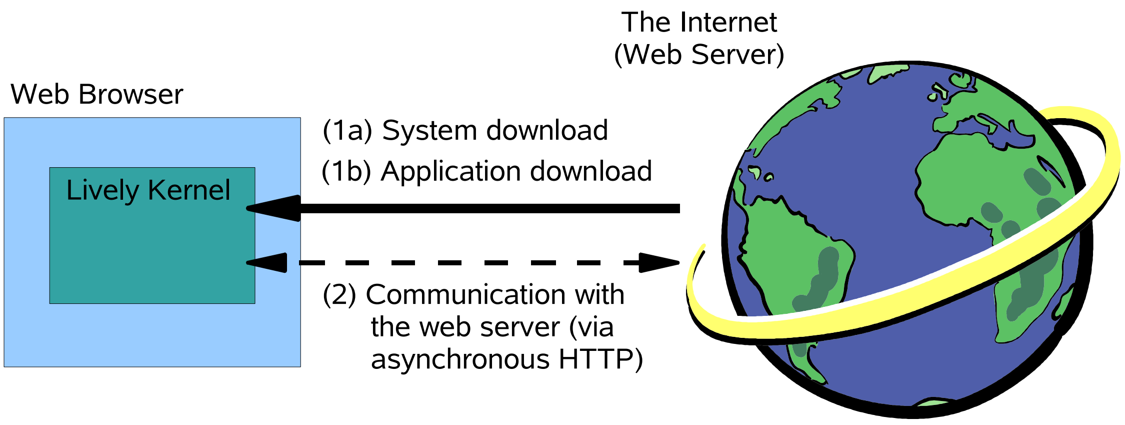


Figure 2: The high-level architecture of the Lively Kernel

Note that from the end user's viewpoint, the Lively Kernel is a visual environment that requires nothing else than an available web browser to run. No external tools, plug-in components or special installation are required. The details related to system downloading or communication with the web server during application execution are entirely transparent to the end user.

2.2 The Lively Kernel Architecture from the Developer's Viewpoint

From the application developer's viewpoint, the Lively Kernel system consists of the following four main components:

1) *The JavaScript programming language and the associated core JavaScript libraries*. We have used the JavaScript programming language as a fundamental building block for the Lively Kernel. A JavaScript engine is available in all the commercial web browsers today. Our current implementation assumes that at least JavaScript version 1.5 is available. The actual version of the JavaScript engine and libraries is dependent on the web browser in which the Lively Kernel is run.

2) *Asynchronous HTTP networking*. All the networking operations in the Lively Kernel are performed asynchronously using the *XMLHttpRequest* feature familiar from Ajax. The use of asynchronous networking is critical so that all the networking requests can be performed in the background without impairing the interactive response of the system. Support for asynchronous HTTP networking is available in all the commercial web browsers today. In the Lively Kernel, the *XMLHttpRequest* interface is commonly used indirectly via the *Ajax.** API of the *Prototype* library (see the introduction of the Prototype library below).

3) *Morphic user interface framework and widgets*. The Lively Kernel is built around a flexible user interface framework called *Morphic*. The Morphic framework consists of about 10,000 lines of uncompressed JavaScript code that is downloaded to the web browser when the Lively Kernel starts. The Morphic user interface framework will be described in more detail in Section 5.

4) *Built-in tools for developing, modifying and deploying applications on the fly*. The Lively Kernel includes built-in tools (such as a JavaScript code browser and object inspector) that can be used for developing, modifying and deploying applications from within the Lively Kernel system itself. These features have been implemented using the reflective capabilities of the JavaScript programming language, and can therefore be used inside the web browser without any external tools or IDEs. The features make it possible, e.g., to write new JavaScript classes, modify or delete existing methods, or change the values of any attribute in the system. Finally, it is possible to export objects or entire web pages, so that the applications written inside the Lively Kernel can also be run as standalone web pages.

Note that the first two components – JavaScript and asynchronous HTTP networking – are part of the web browser, and are therefore provided by the underlying web browser. In contrast, the Morphic UI framework and the development tools associated with it are part of the Lively Kernel files that are loaded into the web browser when the system starts.

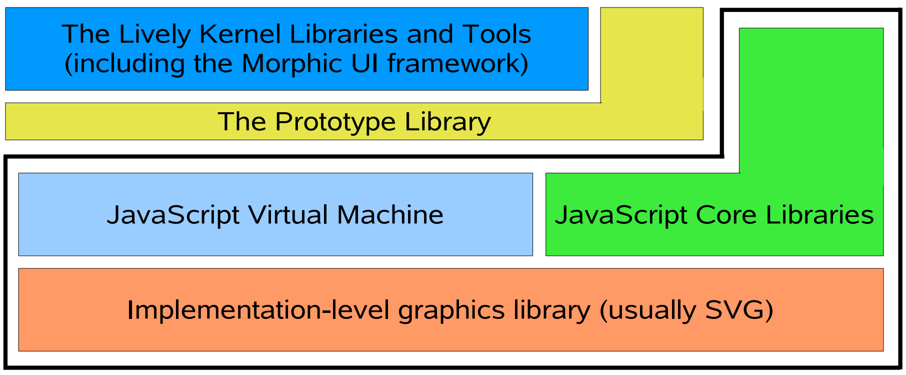

The Lively Kernel Libraries and Tools
(including the Morphic UI framework)
The Prototype Library
JavaScript Virtual Machine
JavaScript Core Libraries
Implementation-level graphics library (usually SVG)

Figure 3: High-level block structure of the Lively Kernel

Figure 3 provides a block structure illustration of the components of the Lively Kernel at runtime. The items within the solid black line represent components that are provided by the web browser, while the rest of the components are usually downloaded from the Web dynamically.

At the bottom of the diagram is the implementation-level SVG graphics library provided by the web browser (see the more detailed description below). The SVG layer is not visible to application developers, except in terms of the functionality that it provides. Next, we have the JavaScript virtual machine and the JavaScript core libraries (the libraries will be discussed in more detail in Section 4). On top of the JavaScript environment we use a JavaScript class library called *Prototype* that provides additional structure to JavaScript applications as well as hides the browser-specific details of the *XMLHttpRequest* networking protocol from the application developer. The Lively Kernel libraries and tools (including the Morphic user interface framework) then reside on top of the JavaScript environment and the Prototype library.

As depicted in Figure 3, the Lively Kernel depends on a number of external libraries. These libraries are summarized in more detail below.

Scalable Vector Graphics (SVG). Our current Lively Kernel implementation depends on an SVG engine that provides the necessary implementation-level support for our graphics architecture, including the affine transformations and matrix operations needed for rotating and scaling objects. SVG is a declarative graphics language and specification designed by the World Wide Web Consortium (W3C). It is supported by most web browsers, although the implementations are still somewhat incompatible or missing altogether from some web browsers. While SVG is commonly used via its HTML-like declarative syntax, the SVG functionality of a web browser can also be accessed programmatically from JavaScript. The Lively Kernel is built on top of the programmatic JavaScript SVG interface.

For further information on SVG, refer to the SVG 1.1 Specification:

http://www.w3.org/TR/2003/REC-SVG11-20030114/

Note that in the context of the Lively Kernel SVG is treated as an *implementation-level API*. We have chosen to use SVG because of its functionality and widespread availability in commercial web browsers. However, to application developers, the Lively Kernel is an environment that is based purely on JavaScript, not SVG. Future releases of the Lively Kernel may include back-ends also for other graphics libraries than SVG.

Prototype library. The Lively Kernel uses a JavaScript class library called *Prototype* (see http://www.prototypejs.org/). In the current version of the Lively Kernel, we use the Prototype library version 1.6.0. The Prototype library provides convenient syntactic mechanisms for defining and extending JavaScript classes. Such mechanisms are missing from the widely used versions of the JavaScript (ECMAScript) language. In addition, the Prototype library introduces a cleaner interface for using asynchronous HTTP networking in a browser-independent, platform-independent fashion. This is important since the implementation of the *XMLHttpRequest* feature still varies from one web browser to another.

3. File Structure and Configuration Options

In this section, we provide a quick overview of the source file structure of the Lively Kernel as well as a summary of the various configuration options. This information is intended primarily for application developers or for those people who want to customize the Lively Kernel to their own needs.

> **Important!** The information in this section pertains to the Lively Kernel version 0.8. The future releases of the Lively Kernel may use an entirely different file structure. For instance, to speed up downloading, we might package all the Lively Kernel functionality in a single, compressed file. Do not assume that the file structure will remain the same in the future.

The table below provides a summary of the source files in the Lively Kernel.

File	Description
index.xhtml	The XHTML file that launches the Lively Kernel and loads all the other files.
prototype.js	The Prototype library that we use as part of the system (http://www.prototypejs.org/).
svgtext-compat.js	Contains code for mapping text support onto the underlying SVG graphics engine.
Core.js	Contains the core Lively Kernel features, the low-level porting interface and the core Morphic user interface framework.
Text.js	Contains all the text-related functionality of the Lively Kernel.
Widgets.js	Contains all the widget definitions for the Morphic UI framework.
Network.js	Contains all the network-related functionality of the Lively Kernel.
Storage.js	Contains all the storage-related functionality of the Lively Kernel.
Tools.js	Contains various tools such as the class browser, object inspector, style editor, etc.
Examples.js	Contains all the sample applications.
Main.js	Contains the JavaScript code that initializes the sample applications.
definitions.svg	Contains embedded resource definitions that are included in the system when it starts.
defaultconfig.js	Defines the default system configuration options for the Lively Kernel.

Note. The Lively Kernel is a "zero-installation" system, i.e., the user does not actually install anything or deal with any of the aforementioned files directly. Instead, the necessary files are loaded automatically by the web browser when the Lively Kernel is started. The loading of the files is performed by the *index.xhtml* file.

In addition to the files listed above, the user may provide a *localconfig.js* file that can override the default system configuration options defined in *defaultconfig.js*. The configuration options are summarized in the table below.

Important! The configuration options are subject to change in the future releases of the Lively Kernel.

Option	**Description**
shiftDragForDup	If true, the user can use shift-dragging to create copies of objects easily.
useNewScheduler	If true, the system uses a newer implementation for script scheduling.
skipMostExamples	If true, most of the example applications will not be loaded upon system startup.
skipAllExamples	If true, none of the example applications will be loaded upon system startup.
showWebStore	If true, a WebDAV file browser application will be launched automatically upon startup.
showCurveExample	If true, a demo application related to curves and paths will be launched upon startup.
loadFromMarkup	If true, some demo applications will be loaded as markup files instead of JavaScript.
showThumbnail	If true, an experimental thumbnail view of LinkMorphs will be used instead of icons.
showNetworkExamples	If false, none of the example applications that require networking will be launched.
ignoreAdvice	If false, ignore function logging through the Prototype.js wrap mechanism.
fakeFontMetrics	If false, try to make up font metrics if the SVG API doesn't work.

In addition to the configuration options listed above, there are a number of additional configuration options related to the sample applications. The options have been defined in the beginning of file *Main.js*. As with the other configuration options, these options are subject to change or may be removed from the future versions of the Lively Kernel.

4. Overview of the Application Programming Interfaces (APIs)

Since the Lively Kernel runs in a web browser and leverages the JavaScript engine that is already part of the browser, the system allows the developers to use all those JavaScript APIs that are provided by the underlying JavaScript environment.

Generally speaking, the Lively Kernel API set consists of the following three APIs:

1) Core JavaScript APIs
2) The Prototype library APIs
3) APIs specific to the Lively Kernel

The Core JavaScript APIs are provided by the web browser's JavaScript implementation. The details related to those APIs are beyond the scope of this document. For a summary of the Core JavaScript APIs, refer to online documentation:

http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference

Also, the following book provides an excellent summary of the Core JavaScript APIs:

David Flanagan, JavaScript: The Definitive Guide (5th edition). O'Reilly Media, 2006.
(Refer to Part III for a summary of the Core JavaScript APIs.)

For a summary of the Prototype library APIs, refer to online documentation:

http://www.prototypejs.org/api

Note that the Prototype library includes an *Ajax.** API that simplifies the use of asynchronous HTTP networking considerably. The Lively Kernel developers are strongly encouraged to use this API instead of the lower level, browser-dependent *XMLHttpRequest* API supported directly by the web browser.

The APIs specific to the Lively Kernel are described in more detail below.

5. The Morphic User Interface Framework

The majority of the JavaScript classes defined by the Lively Kernel are related to a user interface framework called *Morphic*. Morphic is a framework that supports composable graphical objects, along with the machinery required to display and animate these objects, handle user inputs, and manage underlying system resources such as displays, fonts and color maps. Sophisticated built-in mechanisms are provided for object scaling, rotation, object style editing, as well as for defining new user interface themes. The central goal of Morphic is to make it easy to construct and edit interactive graphical objects, both by direct manipulation and from within programs. The Morphic user interface was developed originally for the Self system (http://research.sun.com/self), but it became popular later also as part of the Squeak system (http://www.squeak.org).

The historical references to the original design of the Morphic framework are provided below:

– *Maloney, J.H., Smith, R.B., Directness and liveness in the Morphic user interface construction environment. Proceedings of the 8th annual ACM Symposium on User Interface and Software Technology (UIST), Pittsburgh, Pennsylvania, 1995, pp. 21-28.*

– *Maloney, J.H., Morphic: The Self User Interface Framework. Self 4.0 Release Documentation, Sun Microsystems Laboratories, 1995.*

– *Ingalls, D., Kaehler, T., Maloney, J.H., Wallace, S., Kay, A., Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. Presented at the OOPSLA'97 Conference.Web link: http://ftp.squeak.org/docs/OOPSLA.Squeak.html.*

5.1 The Basic Concepts

The most fundamental concepts in the Morphic user interface framework are *morphs*, *worlds* and *hands*.

Morphs. From the programmer's viewpoint, every visual object in the Lively Kernel is a *morph*. A morph is a visual structure that contains the necessary functionality to manage the coordinates, boundaries and other visual aspects of an object, as well as functions for responding to events and external requests such as moving, grabbing, resizing, rotating, and so on.

Worlds. All the morphs in the Lively Kernel reside in a *world*. A world is a visual container – sort of like a fully interactive, graphical web page or a Wiki – that can be manipulated directly and visually by the users. Before a morph becomes visible, it needs to be added to a world. A world may contain links to other worlds. A world that is contained within another world is commonly referred to as a *subworld*. However, note that the structuring of the worlds does not need to be strictly hierarchical.

Hands. The morphs in a world are manipulated using a *hand*. A hand is the Morphic generalization of the cursor; it can be used to pick up, move, and deposit other morphs, its shape may change to indicate different cursor states, and it is the source of user events in the Lively Kernel architecture. In general, a hand is like a cursor in a desktop operating system, except that in the Lively Kernel a world may have multiple users and therefore contain multiple hands simultaneously. In the current version of the Lively Kernel, multi-user support is not yet enabled.

5.2 Object Composition

The Morphic user interface framework supports a principle known as *object composition/adhesion*: Whenever a morph (visual object) is placed on top of another morph in the user interface, the morph is automatically attached ("glued") to the underlying object. Figure 4 below provides an example of a face-like object that has been created by attaching two red rectangles, a blue polygon and a polyline object onto a green ellipse.

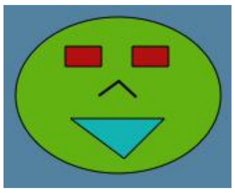


Figure 4: An object created by composition

5.3 Shapes, Colors, Patterns and Affine Transformations

Each morph in the Lively Kernel is associated with a *shape* that defines the underlying graphical attributes of the morph, such as its fill color, border color, border width and opacity (transparency). The basic shape types are summarized in the table below:

Class Name	**Description**
Shape	The base class for defining the underlying shape of a morph
RectShape	A rectangle shape
EllipseShape	An ellipse/circle shape
PolygonShape	A polygon shape
PolylineShape	A polyline shape
PathShape	A path/curve shape

Each shape may be associated with a number of *colors* that are expressed as RGB values. Instead of ordinary colors, various *fill patterns* can be used. The basic fill patterns are summarized in the table below.

Class Name	**Description**
Gradient	The base class for gradient ("sliding") colors
LinearGradient	A linear gradient that changes color from one origin to another
RadialGradient	A radial gradient that changes color from the object's centerpoint to its bounds
StipplePattern	A "criss-cross" color pattern.

Figure 5 provides examples of objects that have been filled with linear gradient or radial gradient colors, respectively.

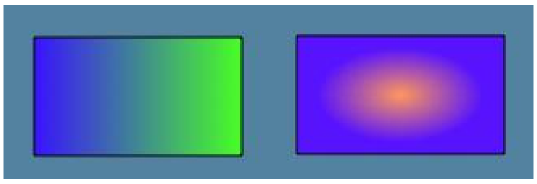
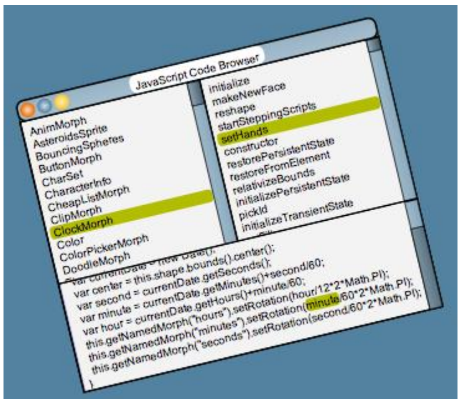


Figure 5: Example of a (horizontal) linear gradient (left) and radial gradient (right)

In addition to its shape, each morph is also associated with a *transform*. A transform is an affine transformation matrix that allows the morph to be repositioned, rescaled or rotated flexibly relative to its current owner. Figure 6 provides an example of a complex morph (a JavaScript code browser) that has been rotated and scaled down (shrunk) using a transformation matrix. Note that all the components (submorphs) within the code browser, such as the *ScrollPane* that displays the source code, have been rotated and shrunk accordingly. All the submorphs can also be transformed (e.g., rotated or scaled) independently of each other, relative to their owner.



```
var currentDate = new Date();
var center = this.shape.bounds().center();
var second = currentDate.getSeconds();
var minute = currentDate.getMinutes()+second/60;
var hour = currentDate.getHours()+minute/60;
this.getNamedMorph("hours").setRotation(hour/12*2*Math.PI);
this.getNamedMorph("minutes").setRotation(minute/60*2*Math.PI);
this.getNamedMorph("seconds").setRotation(second/60*2*Math.PI);
}
```

Figure 6: A JavaScript code browser object that has been rotated and shrunk using an affine transformation

For further information on affine transformations, refer to, e.g.:

http://en.wikipedia.org/wiki/Affine_transformation
http://www.quantdec.com/GIS/affine.htm

Our current Lively Kernel implementation is dependent on affine transformation support and matrix operations provided by the SVG engine. For details on those functions and data structures, refer to:

http://www.w3.org/TR/2003/REC-SVG11-20030114/coords.html#InterfaceSVGMatrix

5.4 Widgets

Once the basic concepts of Morphic have been introduced, including morphs and object composition, it is possible to construct a rich set of widgets that all follow the same basic principles. Below is a summary (JavaScript class hierarchy) of the widgets that are currently supported by the Lively Kernel. All the widgets are morphs, and therefore inherit the basic functionality from class *Morph*. More information about the operations that are available for manipulating morphs will be provided in Section 6. Note that we will not provide a detailed API description of the widgets in this document. For more details on the widgets, refer to file *Widgets.js* and the sample applications.

> **Important!** The widget hierarchy of the Lively Kernel is still under development and may change in the future releases. The future releases are likely to contain more widgets than those listed below.

Class Name	Class Name
Morph	The Morph base class
ButtonMorph	Simple button
ImageButtonMorph	Button with an associated image
ImageMorph	Image object
IconMorph	Image object representing an icon
TextMorph	An object that contains formatted text
CheapListMorph	Simple textual list that supports selection, etc.
MenuMorph	Popup menu
ClipMorph	Clipping view
WindowMorph	Full-fledged window object
TabbedPanelMorph	Tabbed window/panel
TitleBarMorph	An object representing a window title bar
TitleTabMorph	Title bar for a tabbed window
PanelMorph	Simple panel
ScrollPane	Scrollable panel
ListPane	A scrollpane containing a textual list (*CheapListMorph*)
TextPane	A scrollpane that contains text
SliderMorph	Scroll bar object
ColorPickerMorph	Color picker panel
HandMorph	An object representing a hand (cursor) in a Morphic world
HandleMorph	A handle that allows a morph to be resized, rescaled, etc.
SelectionMorph	Selection tray object (for selecting multiple objects)
PasteUpMorph	A visual container onto which other morphs can be pasted
WorldMorph	A Morphic world
LinkMorph	Hyperlink between worlds

Figure 7 contains a few sample widgets to illustrate what kinds of widgets are available and what they commonly look like. In the upper row, there is a *TextMorph*, a *WindowMorph* (containing a *TitleBarMorph*) and a *LinkMorph*. In the lower row, there is a *MenuMorph* (with a *HandMorph* on top of the currently selected menu item), and a *PanelMorph* containing various other morphs such as four *ButtonMorphs*, two *CheapListMorphs*, eight *TextMorphs*, and a *SliderMorph*.

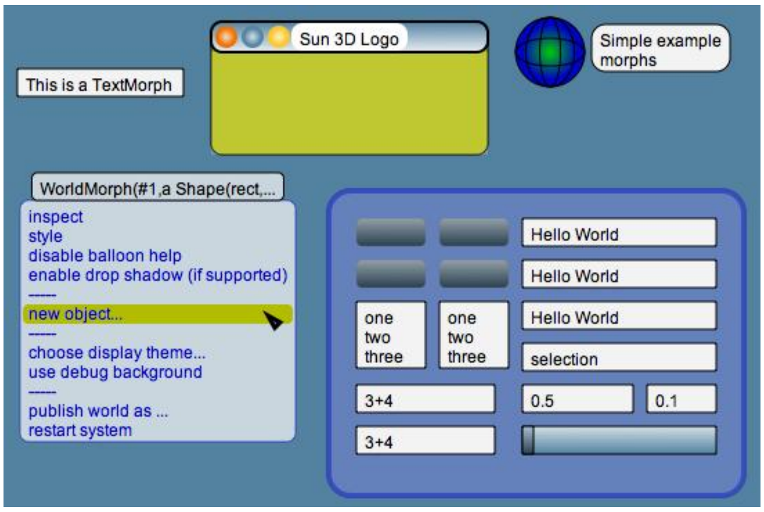


Figure 7: Sample widgets

Note that many of the widgets in the Lively Kernel are intended primarily for system use. For instance, a *HandMorph* represents the user's cursor within a world, and it is not usually manipulated by the programmer, except when changing the shape of the cursor. A *SelectionMorph* (also known as a “selection tray”) allows multiple objects to be selected easily. A *HandleMorph* is used for resizing, reshaping, or rotating other objects. Each *HandleMorph* is presented as a rectangular or round handle (usually a small blue rectangle), and is commonly displayed with balloon help that provides information about the operations that are currently available. Figure 8 shows an example of a *SelectionMorph* (left) and a *HandleMorph* (right) that is used for editing a polygon. A *HandMorph* and balloon help are also visible.

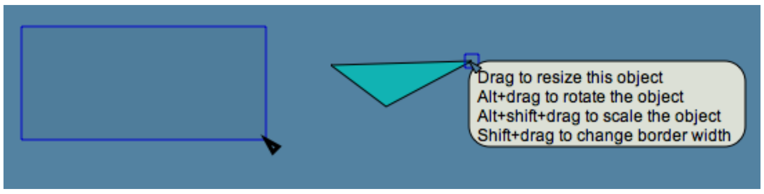

Drag to resize this object
Alt+drag to rotate the object
Alt+shift+drag to scale the object
Shift+drag to change border width

Figure 8: A *SelectionMorph* (left) and a *HandleMorph* (the blue rectangle on the polygon; shown with balloon help)

In addition to the widgets summarized in the class hierarchy above, the Lively Kernel also has a number of tools that have been constructed using the widgets introduced above. These tools include a JavaScript code browser (class *SimpleBrowser*), an object instance inspector (class *SimpleInspector*), and a style panel that allows the visual attributes of objects to be edited easily (class *StylePanel*). All these tools have been defined in file *Tools.js*.

5.5 Styles and Themes

The look and feel of the Morphic user interface is fully customizable, meaning that the widgets may look entirely different in different applications. For instance, a widget such as a *WindowMorph* can look entirely different from the look and feel shown in Figure 7. The UI customization is accomplished using two mechanisms: *styles* and *themes*.

A *style* is a collection of visual attributes associated with an individual morph. These attributes can be changed easily using a style panel (see the *StylePanel* on the left in Figure 9).

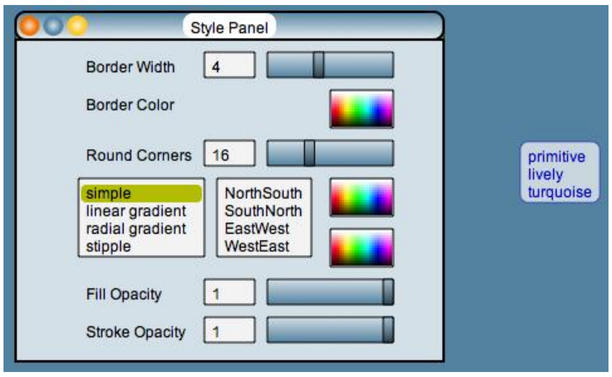


Figure 9: A *StylePanel* (left) and a theme selection menu (right)

A *theme* is a collection of visual attributes associated with a world (*WorldMorph*). When the display theme of a world is changed, the style attributes of all the objects in the world are changed automatically. In the Lively Kernel user interface, themes are selected using a *theme selection menu* (see the menu on the right in Figure 9; in this case, only three display themes are available.) Functions for manipulating themes will be discussed in more detail in Section 7.4.

6. Manipulating Morphs

The morphs in the Lively Kernel can be manipulated in various ways. In this section we provide an overview of the JavaScript APIs that are available for manipulating morphs. For further details on the APIs, refer to the Lively Kernel web site:

http://research.sun.com/projects/lively

The operations for morph manipulation can be categorized broadly as follows:

- instantiating morphs
- copying morphs
- adding and removing submorphs
- manipulating and inspecting the visual attributes of a morph
- moving, resizing, rescaling and reshaping morphs
- obtaining information about the surroundings of a morph
- event handling functionality
- timers and scripting/stepping functionality
- printing and inspecting morphs
- additional functions and tools for morph manipulation

Each of the categories is discussed in more detail below.

6.1 Instantiating Morphs

New morphs are instantiated using the normal JavaScript syntax for object instantiation. The parameters for initialization vary from one morph class to another, but for simple morphs (morphs representing simple shapes such as rectangles or ellipses) they are as follows:

```
var myMorph = new Morph(initialBounds, shapeType);
```

The *initialBounds* parameter determines the location and size of the new morph, and is expressed as a *Rectangle*. The *shapeType* parameter is a string describing the shape of the morph (see the description of shapes earlier in Section 5.3). The possible values of the *shapeType* parameter are “rect”, “rectangle”, “ellipse”, “path”, “polygon”, and “polyline”.

For instance, the following operation would instantiate a morph representing a simple rectangle. The coordinates of the morph relative to its owner is (x, y), and its size is (width, height).

```
var x = 10; var y = 10; var width = 100; var height = 50;
var myMorph = new Morph(pt(x, y).extent(width, height), “rect”);
```

Note. Before a morph becomes visible on the screen, it needs to be added to an *owner* (such as a world) that is currently visible. To add a new morph to the world that is currently active, the following code can be used:

```
var currentWorld = WorldMorph.current();
currentWorld.addMorph(myMorph);
```

6.2 Copying Morphs

Morphs can be copied using the following operation:

```
var myCopy = myMorph.copy();
```

Note that the details related to which morph attributes are copied may vary from one class to another, depending on whether the morph class has overridden the *copy* method or not. By default, a *deep copy* is performed, meaning that all the submorphs, step handlers and scripts associated with the object are copied as well. Note that objects may specifically request not to be copied by defining the following method to return *false*.

```
        myMorph.okToDuplicate()
```

If the *okToDuplicate* method returns *false*, the morph can still be copied programmatically using the *copy* method. However, the "*duplicate*" menu item in the Lively Kernel user interface is disabled/removed, preventing the user from copying the object from the user interface.

6.3 Adding and Removing Submorphs

As was mentioned in Section 5.2, the Morphic user interface framework supports *object composition*. At the implementation level, this means that each morph may have a number of *submorphs* attached to it. Each submorph may, in turn, contain an arbitrary number of submorphs.

The following operations are available for adding submorphs:

```
        myMorph.addMorph(submorph);
        myMorph.addMorphFront(submorph);
        myMorph.addMorphBack(submorph);
        myMorph.addMorphAt(submorph, position);
```

The functions *addMorphFront* and *addMorphBack* add the submorph to the front or back in the display hierarchy (visual layering of objects), respectively. Objects in the front are displayed on top of those objects that are further in the back if the objects overlap visually.

The following operations are available for removing submorphs:

```
        myMorph.removeMorph(submorph);
        myMorph.removeAllMorphs();
        myMorph.remove();
```

The function *removeMorph* removes the specific submorph, while *removeAllMorphs* removes all the submorphs from *myMorph*. The function *remove* removes the morph itself (in this case: *myMorph*) from its owner, and stops all the stepping/scripting functionality associated with the object.

A number of convenience functions are available for submorph manipulation:

```
        var hasSub = myMorph.hasSubmorphs(); // returns true or false
        var topSub = myMorph.topSubmorph(); // returns the topmost submorph
        myMorph.withAllSubmorphsDo(func, args); // applies function func to all submorphs
```

The function *hasSubmorphs* can be used for querying if an object has submorphs or not. The function *topSubmorph* returns the (visually) topmost submorph. The function *withAllSubmorphsDo* automatically applies the given function and arguments to the morph itself and all of its submorphs.

Morphs can also have *named* submorphs. Named submorphs are logically similar to other submorphs, except that they can be accessed by name instead of an index. The following operations are available for setting and getting named submorphs:

```
        myMorph.getNamedMorph("name");
        myMorph.setNamedMorph("name", subMorph);
```

Note. The named submorph mechanism is experimental and may be removed in a future release of the Lively Kernel.

6.4 Manipulating and Inspecting the Visual Attributes of a Morph

There are a large number of operations available for manipulating and inspecting the visual attributes of morphs. Most of this functionality is inherited from class *Visual* that is the parent of class *Morph*. These operations include the following:

```
        myMorph.setFill(fillColor); // Sets the fill color
        myMorph.getFill(); // Gets the fill color
```

```
myMorph.setStroke(borderColor); // Sets the border color
myMorph.getStroke(); // Gets the border color

myMorph.setStrokeWidth(borderWidth); // Sets the border width (in pixels)
myMorph.getStrokeWidth(); // Gets the border width (in pixels)

myMorph.setBorderColor(borderColor); // (duplicate of setStroke)
myMorph.getBorderColor(); // (duplicate of getStroke)

myMorph.setBorderWidth(borderWidth); // (duplicate of setStrokeWidth)
myMorph.getBorderWidth(); // (duplicate of getStrokeWidth)

// Note: Opacity values are numbers between 0...1 (0 = transparent; 1 = opaque)
myMorph.setFillOpacity(fillOpacity); // Sets the fill opacity
myMorph.getFillOpacity(); // Gets the fill opacity

myMorph.setStrokeOpacity(borderOpacity); // Sets the border opacity
myMorph.getStrokeOpacity(); // Gets the border opacity
```

For more information on these operations, refer to the documentation and comments in class *Visual*.

6.5 Moving, Resizing, Rotating and Reshaping Morphs

Morphs can be moved using the following operations:

```
myMorph.setPosition(pt(100, 50)); // Move the morph to position (100, 50)
myMorph.getPosition();            // Get current position as a point

myMorph.translateBy(pt(20, 10)); // Move the down and right by (20, 10) pixels
myMorph.moveBy(pt(20, 10));      // (synonym of translateBy)
```

Note. In order to use these operations, it is necessary to know about classes *Point* and *Rectangle*. All the coordinates within the Lively Kernel are expressed as *Point* object instances. Correspondingly, all the functions that expect parameters as *bounds* or *extents* (x, y, width, height) are expressed using *Rectangle* object instances.

Morphs can be aligned to a certain location using the following operation:

```
myMorph.align(pt1, pt2); // Move the object by pt1-pt2
```

Morphs can be scaled (enlarged or shrunk) relative to their owner using the following operations:

```
myMorph.scaleBy(2); // Double the size of the morph
myMorph.getScale(); // Get the current scale factor (1 = no scaling)
```

Morphs can be rotated relative to their owner using the following operations:

```
myMorph.rotateBy(0.1); // Rotate clockwise by 0.1 radians
myMorph.getRotation(); // Get the current rotation factor (0 = no rotation)
```

Morphs can be reshaped in various ways:

```
myMorph.setShape(newShape); // Change the shape of the morph to newShape
myMorph.setExtent(rect(10, 10, 100, 50)); // Change the bounds of the morph
myMorph.setBounds(rect(10, 10, 100, 50)); // (synonym of setExtent)
myMorph.setVertices([pt(10, 10), pt(20, 20)]); // Change the vertices of a polygon
```

Note. The *setVertices* operation is applicable only to polygon and polyline objects. This operation allows the vertices of a polygon or a polyline to be changed quickly without creating a new morph.

Also note that whenever major changes are done to the shape or layout of an object, the following methods should be called, respectively. The classes that are provided with the Lively Kernel itself perform such change notifications automatically, but user-defined morphs should be prepared to call these methods whenever the shape or layout of a morph changes.

```
myMorph.changed();
myMorph.layoutChanged();
```

6.6 Obtaining Information About the Surroundings of a Morph

Morphs have bindings to various other objects and structures in their environment. For instance, apart from the top-level world, each morph has an *owner* that contains the morph as one of its submorphs. Correspondingly, each morph usually belongs to a *world* and is drawn on a specific *canvas*. The following operations provide information about the surroundings of a morph. Note that the *world* and *canvas* functions return *null* if the morph is not currently attached to any world. For the top level world, the owner is always *null*.

```
myMorph.owner; // Returns the owner of the morph (null if no owner)
myMorph.world(); // Return the world in which this morph is currently located
myMorph.canvas(); // Returns the canvas in which this morph is drawn
```

The following functions are used for obtaining information about the position and the boundaries (extent) of the morph:

```
myMorph.getPosition(); // Returns the current position of the morph
                       // (as a point relative to its owner)
myMorph.getExtent();   // Returns the current extent of the morph (as a rectangle)
myMorph.bounds();      // Returns the full bounding box in owner coordinates
myMorph.innerBounds(); // Returns the bounds of this morph in local coordinates
```

In Morphic, coordinates of objects are expressed relative to their owner. The following operations can be used for translating coordinate values between local coordinates, owner coordinates and world coordinates. Like the names indicate, local coordinates are (x, y) values within an individual object; owner coordinates are (x, y) values that are expressed relative to morph's owner; world coordinates are (x, y) values within the world in which the morph is located.

```
myMorph.worldPoint(localPt); // Convert the given local point to world coordinates
myMorph.localize(worldPt);   // Convert the given world point to local coordinates

myMorph.relativize(ownerPt);  // Convert the given point (expressed in
                              // owner coordinates) to local coordinates
myMorph.relativizeRect(rect); // Convert the given rectangle (expressed in
                              // owner coordinates) to local coordinates
```

The following operations are used for checking whether the given point is inside the morph boundaries:

```
myMorph.containsPoint(ownerPt); // Returns true if the given point (expressed in
                                // owner coordinates) is within the object
myMorph.containsWorldPoint(worldPt); // Returns true if the given point (expressed
                                // (in world coordinates) is within the object
myMorph.fullContainsPoint(ownerPt); // Returns true if the given point is within
                                    // the full boundaries of the object
myMorph.fullContainsWorldPoint(worldPt);
```

6.7 Event Handling

The Lively Kernel is an event-based system that utilizes the same event mechanism as is used by the web browser and the client-side JavaScript APIs in general. This means that each morph can be associated with a number of *event handler functions* that will be called by the system automatically when a certain event occurs, e.g., when a mouse is clicked on a morph.

The following mouse event handler functions are available for all morphs:

```
myMorph.onMouseDown(evt); // Called automatically when the mouse button is
                          // pressed down when the cursor is on this morph
myMorph.onMouseMove(evt); // Called automatically when mouse cursor is moving
                          // on top of this morph
myMorph.onMouseUp(evt);   // Called automatically when the mouse button is
                          // released on top of this morph (after onMouseDown)
myMorph.onMouseOver(evt); // Called automatically when the cursor moves onto
                          // this morph
myMorph.onMouseOut(evt);  // Called automatically when the cursor leaves
                          // this morph
```

The following keyboard event handler functions are available for all morphs (but called only when the morph has keyboard focus):

```
myMorph.onKeyDown(evt);   // Called automatically when a key is pressed down
myMorph.onKeyUp(evt);     // Called automatically when a key is lifted
myMorph.onKeyPress(evt);  // (Combination of the two events above)
```

At the world level (for *WorldMorphs*), the two following additional event handler functions are available:

```
myWorld.onEnter(); // Called automatically upon entering a world
myWorld.onExit(); // Called automatically upon exiting a world
```

Note. All the event handler functions above are *callback functions*. They are not intended to be called by the programmer directly. Rather, the system will call them automatically when necessary.

For each event handler function, an *event parameter* (above: *evt*) will be supplied by the system. The event parameter is generated by the web browser, and it follows general DOM *Event* instance structure of the web browser. For instance, the key code associated with a keyboard event is available as an attribute (*evt.keyCode*) of the event object. Correspondingly, the mouse coordinates for a mouse event are available as DOM attributes *evt.clientX* and *evt.clientY*. For further information on DOM events and the *Event* class, refer to:

David Flanagan, JavaScript: The Definitive Guide (5th edition). O'Reilly Media, 2006.
(Chapter 17: Events and Event Handling)

The Morphic event system has a number of additional operations that control whether a morph is receptive to certain types of events. For instance, the following two methods control whether a morph listens to mouse events and keyboard events, respectively:

```
myMorph.disablePointerEvents();
myMorph.takesKeyboardFocus();
```

The following methods control whether a morph is willing to process drag-and-drop (DnD) requests:

```
myMorph.openDnD(); // Enable dropping onto this morph
myMorph.closeDnD(); // Disable dropping onto this morph
myMorph.toggleDnD(); // Toggle DnD on/off
myMorph.openAllToDnd(); // Enable dropping onto this morph and any of its submorphs
myMorph.closeAllToDnd(); // Disable dropping onto this morph and any of its submorphs
```

Drag-and-drop operations are used internally by the Lively Kernel, e.g., to handle “wormhole” operations to move objects from one world to another when dragging objects onto a *LinkMorph*. However, drag-and-drop functionality can also be used in applications for various user-defined purposes.

6.8 Timers and Scripting/Stepping Functionality

Each morph can have a number of *stepping scripts* (scripts run by timers) associated with it. Stepping scripts are functions that are called automatically by the system at certain intervals. The easiest way to create a stepping script is to use the *startStepping* function.

```
myMorph.startStepping(milliseconds, scriptName, arguments);
```

For instance, the following source code line would create a morph that is rotated by 0.1 radians every 50 milliseconds:

```
myMorph.startStepping(50, “rotateBy”, 0.1);
```

There is an alternative, more object-oriented syntax for creating stepping scripts using the JavaScript object notation.

```
  var action = { actor: this, scriptName: “rotateBy”, argIfAny: 0.1,
                 stepTime: 50, ticks: 0 };
  myMorph.addActiveScript(action);
```

The following operations can be used for creating and controlling stepping scripts:

```
// Call scriptName every 'stepTime' milliseconds
myMorph.startStepping(stepTime, scriptName, arguments);

// The same as 'startStepping' except that script is defined as a function
myMorph.startSteppingFunction(stepTime, function);

 // Stop all stepping scripts associated with this morph
myMorph.stopStepping();
```

```
// Newer syntax for script scheduling (uses special 'action' object syntax)
myMorph.addActiveScript(action)

// Start stepping all the scripts associated with this morph
// This method is called automatically when the morph is added to a world
myMorph.startSteppingScripts();

// Stop stepping all the scripts associated with this morph
// This method is called automatically when the morph is removed from a world
myMorph.stopSteppingScripts();

// Temporarily suspend all the scripts associated with this morph
myMorph.suspendActiveScripts

// Temporarily suspend all the scripts of this morph and its submorphs
myMorph.suspendAllActiveScripts

// Resume all the scripts associated with this morph and its submorphs
myMorph.resumeAllSuspendedScripts
```

6.9 Printing and Inspecting Morphs

Like all the JavaScript objects, morphs can be printed as strings using the following function:

```
myMorph.toString(); // Returns a string containing a textual summary of the object
```

The following function returns a textual representation of a morph using an output syntax that is specific to Morphic:

```
myMorph.inspect();
```

All the morphs in the Lively Kernel can also be printed in JSON (JavaScript Object Notation) format using the following function:

```
myMorph.toJSON(); // JSON: JavaScript Object Notation
```

6.10 Additional Functions and Tools for Morph Manipulation

Using the Morph class as an object factory. In addition to the function categories discussed above, a number of additional functions and tools are available. For instance, the *Morph* class can also behave as a factory for instantiating new, simple morphs such as lines, circles and polygons. More specifically, the following factory methods are available:

```
// Note: in makeLine and makePolygon, vertices is expressed as an array of points
var m1 = Morph.makeLine(vertices, lineWidth, lineColor);
var m2 = Morph.makeCircle(location, radius, lineWidth, lineColor, fillColor);
var m3 = Morph.makePolygon(vertices, lineWidth, lineColor, fillColor);
```

Note that these functions are *class methods*, i.e., they are functions of the *Morph* class itself, rather than methods of its instances.

Attaching popup menu items to morphs. In the Lively Kernel user interface, each morph can be associated with a popup menu that is specific to the morph. The default popup menu can be accessed using the following function:

```
var popupMenu = myMorph.morphMenu();
```

In order to manipulate the items that are visible in the popup menu, you can use the functions of the *MenuMorph* class (see file *Widgets.js*). The functions for manipulating the menu contents include:

```
popupMenu.addItem(["item name", this, 'functionToCall']);
popupMenu.addLine(); // Adds a separator line to the menu
popupMenu.removeItemNamed(“item name”);
popupMenu.replaceItemNamed(“item name”, ["item name", this, 'functionToCall']);
```

In general, items in the menus are expressed as arrays that consist of three elements: a name (the name of the menu item), object (the object to which the menu operation is applied), and the name of the function (within the specified object) to call.

Applying user interface theme definitions to morphs. As was summarized earlier in Section 5.5, the Lively Kernel supports user-defined display themes. A theme is a collection of visual attributes associated with a world (class

WorldMorph). When the display theme of a world is changed, the style attributes of all the morphs in the world are changed automatically. A special *theme specification syntax* (basically a JavaScript object containing the visual attributes) is used for defining new themes. A theme specification can be applied to an individual morph using the following method:

```
        myMorph.applyStyle(themeSpec);
```

The theme specification syntax will be discussed in more detail in a future version of this document.

7. Manipulating Worlds

Worlds are a central concept in the Morphic user interface. A Morphic world captures the notion of an active, dynamic web page. Every morph in the Lively Kernel resides in a world. Before a morph becomes visible, it must be added to a world. When building applications for the Lively Kernel, it is therefore necessary to know about the operations that can be used for creating and manipulating worlds.

Morphic worlds can be thought of like fully interactive, graphical web pages or Wikis. Each world can potentially contain a large number of objects. Numerous applications and widgets can run simultaneously in the same world. However, note that a world can also be a host for an individual, standalone application. For instance, it is quite possible to build conventional, desktop-style applications using the Lively Kernel. In that case, the application will usually consists only of a single world that hosts the entire application.

In this section we provide an overview of the operations that are used for manipulating worlds and subworlds.

7.1 Creating a World

Worlds are instances of class *WorldMorph*. Before a world becomes visible, it must be set current using the operation *setCurrent* and connected to a drawing canvas using the operation *displayWorldOn*, as shown below:

```
        var world = new WorldMorph(Canvas);
        WorldMorph.setCurrent(world);
        world.displayWorldOn(Canvas);
```

After calling the operations above, any morph that is subsequently added to the current world will become visible on the screen automatically.

7.2 Creating Subworlds

The Morphic user interface framework allows an unlimited number of worlds to be created. These worlds do not necessarily have to be linked to each other in any way. However, usually it is most convenient to create subworlds that are organized hierarchically so that the user can navigate in the worlds easily. The easiest way to create hierarchically arranged subworlds is to use class *LinkMorph*.

When a new *LinkMorph* is instantiated, a new subworld is created automatically. A link icon representing the subworld becomes visible in the parent world (the world that owns the subworld). The subworld will also contain a similar link icon that allows the user to navigate back to the parent world. Below are two examples of *LinkMorph* instantiation:

```
        var subworldLink1 = new LinkMorph();
        world.addMorph(subworldLink1); // Attach LinkMorph to the current world
        var subworld1 = subworldLink1.myWorld; // Get pointer to the subworld

        // Position the LinkMorph icon at (60, 460)
        var subworldLink2 = new LinkMorph(null, pt(60, 460));
        world.addMorph(subworldLink2);  // Attach LinkMorph to the current world
        var subworld2 = subworldLink2.myWorld; // Get pointer to the subworld
```

Note that if the first parameter in instantiating a *LinkMorph* is left *null*, an empty subworld will be created automatically. The optional second parameter will indicate the initial position in which the *LinkMorph* icon will be placed (the icon position will initially be the same in the subworld and in the parent world). If no position is specified, a default position (normally the bottom left corner of the screen) will be used.

7.3 Obtaining Information About the Surroundings of a WorldMorph

The following operations can be used for obtaining information about the surroundings of a *WorldMorph*:

```
world.owner; // Returns the owner of this world
world.world(); // Returns the topmost owning world of this world
world.canvas(); // Returns the drawing canvas of the world
```

The *world* operation provides a quick way to access the topmost owning world of a *WorldMorph*, regardless of how deep in the subworld hierarchy the current world is located. Note that for the topmost world in a hierarchy of worlds, *world.owner* is always *null*.

7.4 Other Operations for Manipulating WorldMorphs

Setting and getting the currently displayed world. The following operations can be used for setting and getting the currently active world:

```
var currentWorld = WorldMorph.current()
WorldMorph.setCurrent(someOtherWorld);
```

Note that these operations are *class methods*, i.e., they are operations of the *WorldMorph* class itself rather than methods of its instances.

Also note that you must call the *displayWorldOn* function immediately after calling *setCurrent*, or otherwise the world change will have no impact on the display.

Adding and removing hands. Hands (cursors) can be added and removed using the following operations:

```
var hand = new HandMorph();
world.addHand(hand);
world.removeHand(hand);
```

The first hand in a world can be accessed easily using the following function:

```
var hand1 = world.firstHand();
```

Note that usually the programmer should not manipulate hands directly. Rather, hands are normally created, deleted and handled by the Morphic user interface framework itself.

Attaching popup menu items to worlds. Like all the morphs in the Lively Kernel, each *WorldMorph* can have its own popup menu that can be invoked by alt-clicking the morph (in case of *WorldMorphs*, by Alt-clicking the background of the world). The menu object of a *WorldMorph* can be accessed using the following operation:

```
var popupMenu = world.morphMenu();
```

The menu can be manipulated using the same operations that were summarized earlier in Section 6.10. For additional details, refer to the definition of the *MenuMorph* class (see file *Widgets.js*).

Choosing display themes. As was explained in Section 5.5, the look and feel of the Morphic user interface is entirely customizable and supports user-defined display themes. Themes are associated with *WorldMorphs* and can be changed using the following operation:

```
world.setDisplayTheme(themeObject);
```

Themes are specified using special theme specification syntax (basically, a JavaScript object that defines the values of the various visual attributes). When the *setDisplayTheme* operation is called, the display attribute values of the theme are automatically applied to all the morphs in the world.

As explained earlier, themes associated with a world can also be changed from the user interface. In order to invoke the theme selection menu programmatically, the following function can be called:

```
world.chooseDisplayTheme();
```

Operations related to timers and scripting/stepping functionality. In the Lively Kernel, timers and scripting/stepping functionality is accessed primarily through individual morphs (see Section 6.8). However, there are some useful operations that are available at the world level. Specifically, the following operation opens up a visual inspector that can be used for viewing all the timers that are currently associated with all the morphs in a world:

```
world.inspectScheduledActions(); // Opens an inspector for viewing timed actions
```

Note. This functionality is still subject to change. In the future, we may use a different inspector for timer actions.

8. Sample Applications and Widgets

Application development for the Lively Kernel can occur in two different ways.

1) Applications can be developed interactively within the system itself, using the built-in tools that are part of the Lively Kernel.

2) Applications can be developed using a more conventional, file-based approach, using external tools such as a conventional text editor or a JavaScript IDE.

In this section, we show a number of sample applications in conventional source code format. We start with an overview of the general class definition syntax in the Lively Kernel, followed by three examples. The first two examples implement relatively simple widgets (an analog clock and a JavaScript code browser, respectively). The third example implements a radial engine simulator that can be used for educational purposes to demonstrate how internal combustion engines work.

8.1 Creating New Morph Classes – An Overview

New classes in the Lively Kernel are generally defined using the following syntax:

```
Morph.subclass(“MyCoolMorph”, {

    // Constructor
    initialize: function(/* constructor arguments */) {
        // Initialize instance variables here
        this.foo = 123;
        this.bar = “Hello!”;
        return this;
    },

    myMethod: function() {
        // ... do something ...
    }

});
```

The definition above creates a new class *MyCoolMorph* as a subclass of class *Morph*. The *initialize* operation is the constructor function that is called automatically when creating instances of the class. Note that property definitions (such as the method definitions above) in the class definition are separated by a comma.

Instances of the morph class defined above would be created as follows:

```
var aCoolMorph = new MyCoolMorph(/* constructor arguments */);
```

Additional information on the class definition syntax can be found in the documentation of the Prototype library:

http://www.prototypejs.org/learn/class-inheritance

8.2 Sample Application: ClockMorph Widget

In this section we have included a complete example that defines a widget called *ClockMorph* – a simple analog clock. A *ClockMorph* instance is shown in Figure 10.

The entire source code of the *ClockMorph* class is below.

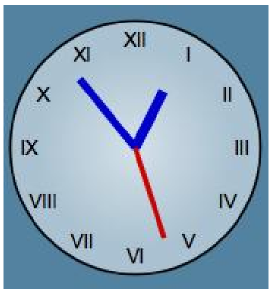


Figure 10: A ClockMorph instance

```
/**
 * @class ClockMorph: A simple analog clock
 */

Morph.subclass("ClockMorph", {

    defaultBorderWidth: 2,
    type: "ClockMorph",

    // Constructor
    initialize: function($super, position, radius) {
        $super(position.asRectangle().expandBy(radius), "ellipse");
        this.openForDragAndDrop = false; // Do not handle drag-and-drop requests
        this.makeNewFace(); // Construct the clock face
        return this;
    },

    // Construct a new clock face
    makeNewFace: function() {

        var bnds = this.shape.bounds();
        var radius = bnds.width/2;
        var labels = [];
        var fontSize = Math.max(Math.floor(0.04 * (bnds.width + bnds.height)),2);
        var labelSize = fontSize; // room to center with default inset

        // Add Roman numerals to the clock
        for (var i = 0; i < 12; i++) {
            var labelPosition = bnds.center().addPt(Point.polar(radius*0.85,
                                    ((i-3)/12)*Math.PI*2)).addXY(labelSize, 0);
            var label = new TextMorph(pt(0,0).extent(pt(labelSize*3,labelSize)),
            ['XII','I','II','III','IV','V','VI','VII','VIII','IX','X','XI'][i]);
            label.setWrapStyle(WrapStyle.SHRINK);
            label.setFontSize(fontSize);    label.setInset(pt(0,0));
            label.setBorderWidth(0);        label.setFill(null);
            label.align(label.bounds().center(),labelPosition.addXY(-2,1));
            this.addMorph(label);
        }

        // Add clock hands
        this.addMorph(this.hourHand = Morph.makeLine([pt(0,0),pt(0,-radius*0.5)],4,Color.blue));
        this.addMorph(this.minuteHand = Morph.makeLine([pt(0,0),pt(0,-radius*0.7)],3,Color.blue));
        this.addMorph(this.secondHand = Morph.makeLine([pt(0,0),pt(0,-radius*0.75)],2,Color.red));
        this.setHands();
        this.changed();
    },

    // Set clock hand angles based on current time
    setHands: function() {
        var now = new Date();
        var second = now.getSeconds();
        var minute = now.getMinutes() + second/60;
        var hour = now.getHours() + minute/60;
        this.hourHand.setRotation(hour/12*2*Math.PI);
        this.minuteHand.setRotation(minute/60*2*Math.PI);
        this.secondHand.setRotation(second/60*2*Math.PI);
    },

    // Will be called when the ClockMorph is placed in a world
    startSteppingScripts: function() {
        this.startStepping(1000, "setHands"); // once per second
    }

});
```

The *ClockMorph* class consists of a constructor, three other methods and a few variable declarations. The method *makeNewFace* creates a new clock face by adding Roman numerals and three hands (representing the hour, minute and second hand) onto the ellipse representing the clock. The method *setHands* is called once a second to update the rotation angle of the hands. The method *startSteppingScripts* initializes the stepping actions, and is called automatically when the *ClockMorph* is placed in a world.

Note the use of *$super* reference in the constructor. The *$super* reference – defined in the Prototype library – allows a method to refer to the corresponding method in its parent class (in this case, the constructor of the parent class *Morph*). For further information on the use of *$super*, refer to the documentation of the Prototype library:

http://www.prototypejs.org/learn/class-inheritance

In order to initialize the *ClockMorph*, the following code can be used:

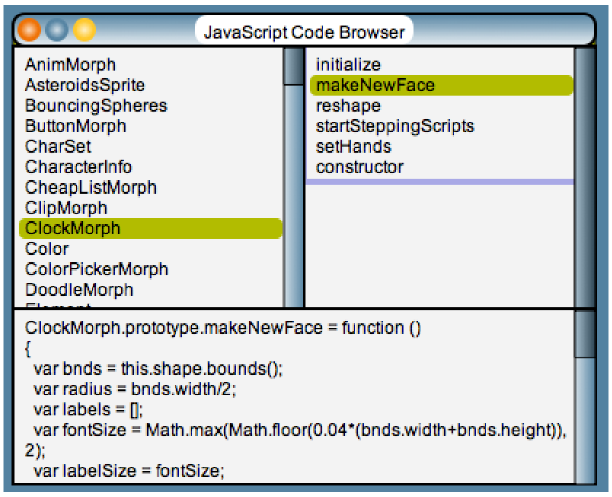
```
    var widget = new ClockMorph(pt(60, 60), 50); // Position (60, 60), radius 50 pixels
    world.addMorph(widget);
    widget.startSteppingScripts();
```

This code creates a new *ClockMorph* instance in position (60, 60), with a radius of 50 pixels.

8.3 Sample Application: JavaScript Code Browser

The source code in this section shows how the JavaScript code browser in the Lively Kernel has been implemented. The class *SimpleBrowser* consists of a constructor and eight methods that utilize the reflective capabilities of the JavaScript programming language that allow the code browser to display the JavaScript classes and methods in the global namespace ("*Global*") of the Lively Kernel itself.

An example of a *SimpleBrowser* instance is shown in Figure 11.



```
ClockMorph.prototype.makeNewFace = function ()
{
 var bnds = this.shape.bounds();
 var radius = bnds.width/2;
 var labels = [];
 var fontSize = Math.max(Math.floor(0.04*(bnds.width+bnds.height)),
2);
 var labelSize = fontSize;
```

Figure 11: A *SimpleBrowser* (JavaScript code browser) instance

The source code of class *SimpleBrowser* is below.

```
/**
 * @class SimpleBrowser: A simple JavaScript class browser
 */

Model.subclass("SimpleBrowser", {

    // Note: Real initialization is done in the 'openIn' method
    initialize: function($super) { $super(); },

    // Get the list of all the classes in the system
    getClassList: function() {
        return Class.listClassNames(Global).sort();
    },

    setClassName: function(n) { this.className = n; this.changed("getMethodList"); },

    // Get all the methods associated with 'className'
    getMethodList: function() {
        if (this.className == null) return [];
        else {
            if (this.className == 'Global')
                 return Global.constructor.functionNames().without(this.className);
            else return Global[this.className].localFunctionNames();
        }
    },

    setMethodName: function(n) { this.methodName = n; this.changed("getMethodString"); },

    // Get the method code associated with 'className' and 'methodName'
    getMethodString: function() {
        if (!this.className || !this.methodName) return "no code";
        else return Function.methodString(this.className, this.methodName);
    },

    // Override previous method definition with eval()
    setMethodString: function(newDef) { eval(newDef); },
    // Open a code browser in the given location
    openIn: function(world, location) {
        world.addMorphAt(new WindowMorph(this.buildView(pt(400,300)),
                         'JavaScript Code Browser'), location);
        this.changed('getClassList')
    },

    // Build a three-pane panel for the code browser
    buildView: function(extent) {
        var panel = PanelMorph.makePanedPanel(extent, [
            ['leftPane', ListPane, new Rectangle(0, 0, 0.5, 0.6)],
            ['rightPane', ListPane, new Rectangle(0.5, 0, 0.5, 0.6)],
            ['bottomPane', TextPane, new Rectangle(0, 0.6, 1, 0.4)]
        ]);

        // Connect panes with specific methods in the SimpleBrowser class
        var m = panel.getNamedMorph('leftPane');
        m.connectModel({model: this, getList: "getClassList", setSelection: "setClassName"});
        m = panel.getNamedMorph('rightPane');
        m.connectModel({model: this, getList: "getMethodList", setSelection: "setMethodName"});
        m = panel.getNamedMorph('bottomPane');
        m.connectModel({model: this, getText: "getMethodString", setText: "setMethodString"});

        return panel;
    }

});
```

Class *SimpleBrowser* inherits from the Lively Kernel class *Model* that provides a straightforward *Observer* (publish-subscribe) pattern implementation (see http://en.wikipedia.org/wiki/Observer_pattern) to manage dependencies between different objects. In this case, the *Model* dependency mechanism is used for hooking up the three panes in the user interface of the code browser to specific methods of the *SimpleBrowser* class. For instance, when the user performs a selection in the class panel (top left panel in a a *SimpleBrowser* instance), the method *setClassName* will be called automatically (see the *buildView* method code above).

8.4 Sample Application: Radial Engine Simulator

In this section we have included a more comprehensive example application: a radial engine simulator. This application can be used for educational purposes to illustrate how radial engines (and more generally, internal combustion engines) work. For a general introduction to radial engines, refer to, e.g., http://en.wikipedia.org/wiki/Radial_engine.

The radial engine simulator consists of a *WindowMorph* that contains a central crankshaft and a number of cylinders arranged radially (see Figure 12). All the components of the engine are created as morphs in the application. The different colors in the cylinders indicate the four different cycles/strokes in the engine (intake stroke, compression stroke, power stroke, exhaust stroke). A number of *MenuMorphs* are used for controlling the number of cylinders, the ignition timing sequence, and the general state of the simulation (running, stopped, manual stepping).

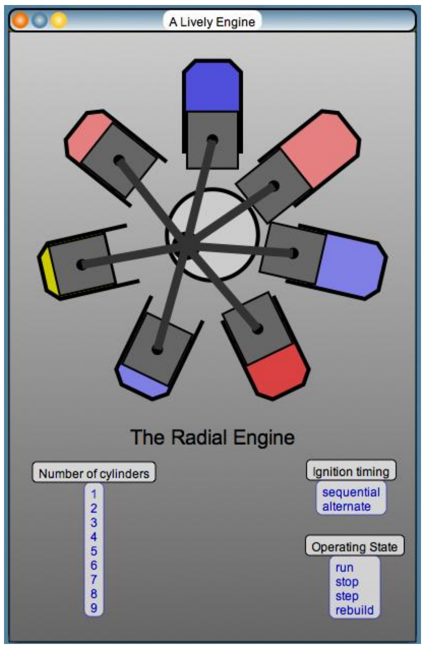


Figure 12: The Radial Engine Simulator

The source code of the radial engine simulator (class *EngineMorph*) is included below.

```
/**
 * @class EngineMorph: The Radial Engine demo
 */

Morph.subclass(“EngineMorph”, {

    initialize: function($super, fullRect) {
        $super(fullRect, "rect");
        this.setFill(new LinearGradient(Color.gray,
                         Color.darkGray, LinearGradient.NorthSouth));
        this.makeLayout();
        this.running = true;
    },

    // Create the general layout of the demo (crankshaft and the menus)
    makeLayout: function() {
        var bnds = this.innerBounds().withHeight(this.innerBounds().width);
        var center = bnds.center();
        this.stroke = bnds.height*0.14;
        this.alternateTiming = false;
        this.crank = Morph.makeCircle(center, this.stroke*0.8, 4,
                                      Color.black, Color.gray);
        this.addMorph(this.crank);
        this.crankPin = Morph.makeCircle(pt(0, -this.stroke/2), this.stroke*0.25,
                                         0, null, Color.black);
        this.crank.addMorph(this.crankPin);
        this.crankAngle = 0;  // goes up to 4*pi, while rotation wraps at 2*pi
        this.angleStep = Math.PI/8;

        var menu = new MenuMorph([]);

        for (var i=1; i<=9; i++) menu.addItem([i.toString(), this, 'makeCylinders', i]);
        menu.openIn(this, pt(80,440), true, "Number of cylinders");

        menu = new MenuMorph([
            ["sequential", this, 'setAlternateTiming', false],
            ["alternate", this, 'setAlternateTiming', true]
        ]);
        menu.openIn(this, pt(300,440), true, "Ignition timing");

        menu = new MenuMorph([
            ["run", this, 'setRunning', true],
            ["stop", this, 'setRunning', false],
            ["step", this, 'doStep'],
            ["rebuild", this, 'rebuild']
        ]);
        menu.openIn(this, pt(315,515), true, "Operating State");

        var label = new TextMorph(new Rectangle(0, 0, 100, 20),
                                  "The Radial Engine").beLabel();
        label.setFontSize(20);  this.addMorph(label);
        label.align(label.bounds().topCenter(), bnds.bottomCenter().addXY(0, -20));
    },

    // Construct the given number of cylinders
    makeCylinders: function(nCylinders) {
        // Build cylinder-piston assembly with center or rotation at crank center
        this.crankAngle = 0;
        this.crank.setRotation(this.crankAngle);
        var bnds = this.innerBounds().withHeight(this.innerBounds().width);
        var center = bnds.center();
        var relBore = 0.14;
        var cr = bnds.scaleByRect(new Rectangle(0.5 - (relBore/2), 0.1, relBore, 0.2));
        var dHead = cr.width*0.2;  // slight dome at top of cylinder -- room for valves
        var cylVerts = [cr.topRight(), cr.bottomRight(),  //vertices of cylinder polygon
            cr.topRight().addXY(0, this.stroke), cr.topLeft().addXY(0, this.stroke),
            cr.bottomLeft(), cr.topLeft(),
            cr.topLeft().addXY(dHead, -dHead), cr.topRight().addXY(-dHead, -dHead),
            cr.topRight()
        ];
        cylVerts = Shape.translateVerticesBy(cylVerts,
                         this.crank.bounds().center().negated());
        var cylinder = Morph.makePolygon(cylVerts, 4, Color.black, Color.gray);
        cylinder.setPosition(cr.topLeft().addXY(0, -dHead));
        var pistonBW = 2;
        var pistonDx = (cylinder.getBorderWidth() + pistonBW) / 2;
        var piston = new Morph(cr.insetByPt(pt(pistonDx, (cr.height-this.stroke)/2)),
                               "rectangle");
        piston.setFill(Color.darkGray);
        piston.setBorderWidth(pistonBW);
        cylinder.addMorph(piston);
        var wristPin = Morph.makeCircle(piston.innerBounds().center(), cr.width*0.1,
                                        0, null, Color.black);
        piston.addMorph(wristPin);

        // Duplicate and rotate the cylinder assembly to complete the engine
        if (this.cylinders) this.cylinders.each( // remove any previous assemblies
            function(each) { each.connectingRod.remove(); each.remove(); }
        );
        this.cylinders = [];
        for (var i=0; i<nCylinders; i++) {
            var cyl = cylinder.copy();
            this.addMorph(cyl)
            cyl.angle = (Math.PI*2/nCylinders)*i;
            if (this.alternateTiming && i%2 == 1) cyl.angle += Math.PI*2;
            cyl.setRotation(cyl.angle);
            cyl.piston = cyl.topSubmorph();
            cyl.piston.topPos = cyl.innerBounds().topLeft().addXY(pistonDx, dHead);
            cyl.wristPin = cyl.piston.topSubmorph();
            this.movePiston(cyl);
            this.cylinders.push(cyl);
            cyl.connectingRod = Morph.makeLine(
                [this.localizePointFrom(cyl.wristPin.bounds().center(), cyl.piston),
                this.localizePointFrom(this.crankPin.bounds().center(), this.crank)],
                cr.width*0.15, Color.gray.darker(2)
            );
            this.addMorph(cyl.connectingRod)
        };
    },

    // Move a piston according to the four engine cycles/strokes
    movePiston: function(cyl) { // Method to move piston and connecting rod
        var pi = Math.PI;
        var phase = (this.crankAngle - cyl.angle);
        if (phase < 0) phase += pi*4;
        var dy = (Math.cos(phase) - 1) * this.stroke/2;
        cyl.piston.setPosition(cyl.piston.topPos.addXY(0, -dy));
        var cycle = Math.floor(phase / pi);
        var frac = phase / pi - cycle;  // fractional part of cycle used to mix colors
        switch (cycle) {
            case 0: cyl.setFill(Color.blue.lighter());
                    break;  // intake
            case 1: cyl.setFill(Color.blue.mixedWith(Color.blue.lighter(), frac));
                    break;  // compression
            case 2: cyl.setFill(Color.red.lighter().mixedWith(Color.red, frac));
                    break;  // power
            case 3: cyl.setFill(Color.red.lighter());
                    break;  // exhaust
        }
        // ignition
        if (Math.abs(phase-2*pi) < this.angleStep/2) cyl.setFill(Color.yellow);
    },

    setRunning: function(trueOrFalse) { this.running = trueOrFalse; },

    nextStep: function() { if (this.running) this.doStep(); },

    doStep: function() {
        this.crankAngle += this.angleStep;
        if (this.crankAngle > Math.PI*4) this.crankAngle -= Math.PI*4;
        this.crank.setRotation(this.crankAngle);  // Rotate the crankshaft
        this.cylinders.each(function(cyl) {
            this.movePiston(cyl);  // Move the pistons
            cyl.connectingRod.setVertices(  // Relocate the connecting rods
                [cyl.connectingRod.localizePointFrom(cyl.wristPin.bounds().center(),
                 cyl.piston),
                 cyl.connectingRod.localizePointFrom(this.crankPin.bounds().center(),
                 this.crank)] );
        }.bind(this) );
    },

    setAlternateTiming: function(trueOrFalse) {
        // Demonstrate alternate and sequential firing order
        this.alternateTiming = trueOrFalse;
        this.makeCylinders(this.cylinders.length);
    },

    rebuild: function() {
        this.removeAllMorphs();
        this.makeLayout();
        this.makeCylinders(this.cylinders.length);
    },

    startSteppingScripts: function() { this.startStepping(100,'nextStep'); }

});
```

The main functionality of the radial engine simulator is defined in four functions. Function *makeLayout* creates the general layout of the demo (the crankshaft and the menus for controlling the demo). Function *makeCylinder* constructs the given number of cylinder (the number of cylinders can vary from 1 to 9). The function *movePiston* moves the piston of the given cylinder according to the four different strokes of the cylinder. The function *doStep* runs the demo one step forward. The *doStep* function is called periodically as a stepping script (from function *nextStep*) every 100 milliseconds, unless the user has stopped the engine.

9. Advanced Topics for Future Discussion

There are a number of advanced topics that have not been covered in this document. Such topics include the following.

Manipulating text. The Morphic user interface framework implements its own text handling functions that are defined in class *TextMorph* (file *Text.js*). *TextMorph* functionality will be covered in a future version of this document.

Using the *Model* and *SimpleModel* classes. The Morphic user interface framework includes an Observer (publish-subscribe) pattern implementation (see http://en.wikipedia.org/wiki/Observer_pattern) that allows the programmer to manage dependencies between different objects easily. Classes *Model* and *SimpleModel* will be covered in more detail in a future version of this document.

Defining new user interface themes. As was described earlier in this document, the Morphic user interface framework supports user-defined display themes. The syntax for specifying new themes will be covered in a future version of this document.

Using the built-in tools for application development. The Lively Kernel differs from most other web application development systems in the sense that in the Lively Kernel application development can occur inside the system itself, using tools that have been built into the Lively Kernel itself. No external tools are needed, apart from the web browser in which the Lively Kernel runs. The built-in development tools will be covered in more detail in a future version of this document.

Saving, importing and exporting Lively Kernel objects and worlds. Objects and classes created within the Lively Kernel can be saved, imported and exported in different ways. These mechanisms will be covered in more detail in a future version of this document.

Porting the Lively Kernel onto additional target platforms. We have ported the Lively Kernel to run on a number of alternative target platforms. Our porting experiences and porting advice will be summarized in the future versions of this document.