MorphR – a Morphic GUI in Ruby

Kerstin Lyngfelt
MorphR – a Morphic GUI in Ruby

Kerstin Lyngfelt

Examiner:
Lektor Robert Feldt

Supervisor:
Richard Torkar

Trollhättan, 2004
2004:PM03
Sammanfattning

Grafiska användargränssnitt har alltid varit en utmaning att bygga för programvaruutvecklare. Design mönster har utvecklats för att hjälpa till i det arbetet och ett av dessa mönster är Model-View-Controller (MVC). Även om MVC är det mest använda designmönstret vid användargränssnitts utveckling kan MVC anses lite stelt och har vissa begränsningar. Därför har modifierade versioner av MVC använts vid utvecklandet av grafiska ramverk. Vi kommer att titta på både MVC mönstret och på hur bland andra Morphic, Ion och AppKit har designat sina ramverk utifrån MVC. Målet med analysen är att designa och implementera ett modernt, flexibelt vektorbaserat ramverk i Ruby och använda OpenGL som grafikhanterare. Vi kommer också att titta på hur man kan använda teman för att olika grafiska presentationer av en applikation.
# Innehållsförteckning

Sammanfattning .............................................................................................................. ii

1. Introduction .................................................................................................................4

2. Background .................................................................................................................5
   2.1. Ruby ...........................................................................................................................................................5
   2.2. OpenGL ......................................................................................................................................................5
   2.3. Themes .......................................................................................................................................................6
   2.4. Model-View-Controller ..............................................................................................................................6
   2.5. Morphic ......................................................................................................................................................6
   2.6. Ion...............................................................................................................................................................7
   2.7. Avalon and XAML .....................................................................................................................................7

3. Method ........................................................................................................................8

4. Analysis of GUI design alternatives............................................................................8
   4.1. Design of MVC ..........................................................................................................................................8
   4.2. Design of Morphic......................................................................................................................................8
   4.3. Design of AppKit........................................................................................................................................9
   4.4. Design of Ion ..............................................................................................................................................9

5. Design of MorphR.....................................................................................................10

6. Results .......................................................................................................................11
   6.1. Design.......................................................................................................................................................11
   6.2. Morphs......................................................................................................................................................11
   6.3. Timers.......................................................................................................................................................11
   6.4. Themes.....................................................................................................................................................11

7. Discussion .................................................................................................................12

8. Future Work ..............................................................................................................12

9. Conclusion...............................................................................................................12

10. Acknowledgements.................................................................................................12

11. References...............................................................................................................12
Abstract

Building Graphical User Interfaces (GUI) has always been a challenge for software engineers. Many patterns have been developed to help software engineers in their work, some patterns with longer lifetime than others. One pattern that has been used since the late 1970s is the Model-View-Controller (MVC). The MVC pattern has limitations and therefore modifications to the MVC pattern have been made to suit different purposes. In this study, which aims to design and implement a modern, flexible, easily extendible vector-based GUI for a dynamic programming language, both the MVC pattern and other design options like Morphic and Ion were analyzed. This paper initially looks at how MVC has been implemented in different GUI frameworks and furthermore takes the first step in applying said design in a flexible implementation in Ruby. It will also look at how to use themes in order to provide different graphical options of the same application.

1. Introduction

Mankind has been communicating through pictures long before writing existed [1]. A picture does not have to be related to language or country, but is international as few other communication forms. Since communication and interaction between countries have increased, due to improved global communications networks, graphics is an important source of information.

Even though computer applications have been around for decades, a GUI is still hard to build in an easy and unified way. For instance standards and recommendations are not suited for every purpose and there are almost as many recommendations as there are developers. Design patterns are one way to streamline software and reuse experience. The Model-View-Controller (MVC) pattern is one of the most used patterns in GUI frameworks.

GUIs have always been hard to implement due to the complexity [2] when text and pictures are used together to present and visualize the application to the users. There is also interaction from the users through the mouse and the keyboard that has to be handled by the GUI.

Technology changes rapidly and with no signs of a slow down, Moore’s law [3] is still valid. GUIs can use more powerful technology, have increased capacity and contain more functionality than earlier, which creates increased GUI requirements from the users at the same pace as the technology development.

Despite all problems, a picture is still better suited to cross language barriers, so there is much gain in finding a method that works for its purpose when it comes to GUIs. There have been a number of recommendations on how a user interface should be designed and implemented.

A GUI is easier to understand if the user is familiar to the look and feel of it [4]. One way to customize a GUI is to use themes like in e.g. the Linux window manager KDE [5]. Themes are a way to give a united impression of the components in an application with coloring and shaping.

When it comes to handling graphics, different approaches are to be considered. In this paper focus has been on a few different types of GUI approaches. The traditional Model-View-Controller (MVC), a design pattern that separates the model from the view and control of it [6] is often used as a base while building GUIs. Morphic [7], the graphical framework of Squeak [8] uses a slightly different implementation of the MVC pattern, while Ion implements the MVC pattern using OpenGL as back-end.

This paper aims to design and implement a flexible GUI framework, called MorphR, by using the high-level programming language Ruby [9] combined with the low-level graphical back-end OpenGL [10].
2. Background

In order to provide flexibility to the implementation of the MorphR GUI framework the high-level, object-oriented programming language Ruby is used together with low-level instructions through OpenGL. This combination makes it possible to write high-level code at yet have access to the low-level drawing instructions. Used together with themes, a GUI can be customized to suite different user requirements.

In the late 1970s a design pattern called Model-View-Controller [11] was developed to simplify and visualize the users conceptual model of the application and also to simplify development and maintenance of GUIs. The ideas of MVC were used in the object-oriented programming language Smalltalk–80 [12]. According to Bosch et al. [13], MVC was the first widely used pattern for building a framework. They describe a framework as a set of classes that together works as a reusable design to solve problems or even construct a new system and that is how frameworks are used in this paper.

Morphic [7] is a graphical framework in the Smalltalk based project Squeak [8], based on a user interface construction environment for the Self-4.0 system [14]. Self-4.0 is similar to Smalltalk, but is prototype-based and without classes and assignments. Morphic uses a modified version of MVC, with the view and controller roles combined, and with the Smalltalk-based graphical backend BitBlt [7].

A more standard graphical backend is OpenGL [10], which contain a number of functions to control and write graphics. OpenGL is platform independent and can be accessed from a number of high-level languages.

Another graphical framework is Io’s graphical interface Ion [15]. Ion is based on a similar modified version of MVC as Morphic, but uses OpenGL as graphical backend. Ion is influenced by NeXT’s AppKit [16], which also has influenced GnuStep [17], an open source implementation of AppKit.

Many 2D GUI toolkits uses a widget-based architecture [18]. This means that the component, for example a button, is implemented by a set of features and behaviors that are the same for all buttons. These kinds of widgets tend to be static in the way that the features are set once and are not easily changed, which can make the components inflexible [19]. Lecolinet [18] solves this problem by combining the widget-based architecture with the scene-graph approach where the scenes are modeled by a graph of nodes representing properties and behaviors of the scene.

This combination is providing stability and reliability from the widget architectures and the flexibility of the scene-graph approach.

2.1. Ruby

Ruby [9] is an object-oriented programming language with its roots in script languages like Perl and Python, but with the strength of a programming language like Smalltalk. Ruby is portable to most platforms, which makes it flexible.

GUI alternatives in Ruby are Tcl/Tk, GTK, FXRuby to mention a few. Tk is using a binding to Ruby, which is close to the Perl binding so Perl programmers would probably feel confident in using the Tk with Ruby. FXRuby is coming from FOX [20] and provides the most common GUI functionality. The component-based programming language Cocoa that runs on Mac OS has an implementation of Ruby in the RubyCocoa [21] distribution with Objective-C and AppKit, but this distribution is Macintosh specific and thus not platform independent.

2.2. OpenGL

Computer hardware and software have increased their performance and so has also the graphic applications. Therefore a need for a quick, easy and portable application programmer interface (API) has become essential.

OpenGL came from GL, which was developed for Silicon Graphics Incorporated (SGI) hardware. GL was simple, but very powerful and became the basis for OpenGL. Now OpenGL contains more than 200 functions and is portable to most computer systems. It has a well defined interface and any system that supports that interface can benefit from OpenGL advantages [10].

Since OpenGL is platform independent and programs need to interact with the operating system, OpenGL uses a toolkit called OpenGL Utility Toolkit (GLUT). This toolkit supports standard operations of the most common windowing systems today.

The over 200 functions included in OpenGL can be grouped as primitive functions, attribute functions, viewing functions, input functions and control functions. There are also OpenGL extensions, which are not included in the standard distribution of OpenGL, but can be useful in special applications.

The most popular version of OpenGL is the C binding and contains two libraries called gl and glu. Since the C programming language does not allow overloading as in object-oriented languages, there are
multiple forms of the same functions. OpenGL uses the standard types of C, but have their own types as well based on the standard types. Differences in notation between different high-level languages may occur, but the same OpenGL functions are executed.

Due to its simplicity, and yet powerfulness, OpenGL is used in many different fields and applications. Döllner and Hinrichs [22] have developed a generic rendering system called VRS, where OpenGL is wrapped together with other systems under C++. Elsherbeni et al. [23] are using OpenGL together with MS Visual Basic to present an interactive GUI for electromagnetic and antenna applications. OpenGL is also used to rapidly display large images with a tool, which uses a combination of different technologies such as Python, C, GTK and OpenGL [24].

2.3. Themes

In this paper themes are used to provide alternative presentation settings in order to be able to personalize the appearance of an application. Depending on what theme is selected, all components have a uniform coloring and shaping. Themes can either be provided by bitmap images or vector-based drawing instructions depending on the structure of the application.

2.4. Model-View-Controller

Model-View-Controller (MVC) is a design pattern that can be used to design computer applications. One of the central ideas of MVC is to separate data, presentation and interaction from each other in order to handle multiple views to present the same data [25]. This separation of responsibilities provides large flexibility in presenting, controlling and maintaining data. MVC divides a GUI application in models, views and controllers where the models are the domain representatives, i.e. objects or data, of the application, views are presenting the models and controllers handle the views events [26]. With this division, the impact of user interface changes minimizes and the reusability is increased.

Speaking in terms of design patterns [27], MVC can be considered as a combination of three other patterns. The views can be seen as the Observers of models, views are often Composites and controllers define a Strategy for the behavior of views [25]. When a model changes, often due to a call from the controller, the view is notified and changes its presentation according to the models change.

As an example the model can consist of statistical data. The view can be either a graphical diagram or a spreadsheet or a graph representing the same data [28]. One view can also represent data from more than one model. The view and the controller are more tightly coupled than views and models and therefore there are implementations that are based on MVC, but have merged the view and controller into one component.

2.5. Morphic

The word morph comes from Greek and can be translated to shape or form. The word morph can also be used as a verb and have the meaning to change shape or form over time.

All components in Morphic, i.e. windows and buttons, are called morphs and each morph can be a parent or a child to other morphs. Morphic is using the terms supermorph and submorph for the parent and the child. With this approach a window can have submorphs like buttons and text fields and a button in that window can have an image or a label as its submorph.

The Morphic framework was originally developed at Sun Microsystems for the Self 4.0 system and then rewritten from scratch in Smalltalk for Squeak [7]. Morphic is based on the MVC pattern, but with the roles of view and controller combined by handling both user input and display. The purpose of the framework is to simplify the building of easy-to-use and interactive user interfaces.

The model is not needed in many morphs since they are stand-alone graphical objects. Where the morph needs a model it can be its own model, for example the StringMorph holds its own string instead of having a separate model class for that. MVC is supported by Morphic in the way it allows multiple views represent the same model and uses the same update mechanism to inform views about model changes.

The Morphic modification of the MVC pattern requires more computer resources than Smalltalk’s original implementation of MVC [7]. When MVC was developed, performance was an important issue, but since capacity of hardware has increased without raising the cost, performance is not such a big issue anymore. The extended resources give Morphic the option to add more interactivity that gives a sense of directness and liveness to the framework. Everything is happening in the here and now. In the Smalltalk’s MVC pattern, the only view that could draw on the display was the top view, for example the window, and only in its own bounds. In Morphic, all views are updated continuously by incremental screen updating [7]. A clock morph created from a label morph
MorphR – A Morphic GUI in Ruby

Morphic has a feeling of solidity or concreteness while handling the feedback of resizing or moving an object. In Morphic it appears as though real objects remain solid even if moved around or partly overdrawn by another object. Conversely in Smalltalk’s original implementation of the MVC pattern only a wire frame object is shown while moving an object. This option is less resource consuming than Morphic, suited for computers with limited resources. Morphic therefore provides an outline-only window dragging and resizing as an option for slower machines.

The Morph class contains default implementations for all the aspects of behavior of the morph thus making it easy to implement an empty subclass of Morph and add functionality and behavior incrementally. The testing is also simplified with this method since each behavior can be tested one by one.

The visual appearance of a morph is intended to give the illusion of a tangible object that can be picked up and altered after the users needs. The overlapping functionality creates a sense of depth that Maloney [7] calls “two and a half dimension” encouraged by a drop shadow when a morph is to be manipulated. New kinds of morphs can be created incrementally by dragging new morphs over the existing and drop them at the required place.

![Figure 1. A star in Morphic](image)

As shown in Figure 1, a drawn object can be picked up and moved by the mouse. The left star, a), is drawn and placed in the window and the middle star, b), is picked up by the mouse and can be moved around in the window. The shadow adds the illusion that it is a physical object that is moved around. In the right picture, c), the star is shown with a frame of tools for altering the appearance of the morph. This tool frame is called a *halo* and each tools function is shown as a balloon, if moving the mouse over the halo tool.

A feature that separates Morphic from many other graphical frameworks is its liveness. This liveness is possible due to a timer with a stepping mechanism. The timer sets the step frequency and all morphs can implement the step message and set a timer with the frequency. This mechanisms together with the drag and drop support gives the desired liveness impression that the developers of Morphic wanted.

Since Morph is a concrete class and not an interface, subclasses to Morph only needs to implement methods that differ from the Morphs standard implementations, thus making implementations very flexible. Default methods are always available while new can be developed if needed. The most used drawing engine in Morphic is FormCanvas, which is a subclass of Canvas. FormCanvas uses BitBlt, which is a Smalltalk implementation for drawing, but it is also possible to send graphics commands over a socket on a network or Postscript commands to a printer, if creating other subclasses to Canvas [7].

According to Maloney [7], the original design decision to have a global static coordinate system will probably be reconsidered in a near future. Maloney suggests that each morph should have positioning relative to its supermorph with optional rotation and scaling instead of today’s system where the Morph class is controlling those features.

### 2.6. Ion

Io [15] is a small, prototype-based programming language, mostly inspired from Smalltalk [12], Self [14] and NewtonScript [30]. Io is open sourced under a BSD [31] license. The graphics of Io is created by the Ion framework, which uses OpenGL for handling graphics and events. Ion uses a similar modified version of MVC as Morphic and is, according to the author of Io Steve Dekorte, influenced by NeXT’s AppKit [16]. The goal with Io is to be a small, simple language where everything is modifiable in runtime. It is still early in its development, but the design ideas are worth looking at in this paper. Ion has many similarities with Morphic, particular in the design, even if Ion is much smaller than Morphic at the time of writing this paper.

Ion is currently using themes by providing different bitmap images to represent the different components. This approach may be inflexible when it comes to resizing and combining two different components.

### 2.7. Avalon and XAML

Currently Microsoft is developing the next generation of Windows, also known as Longhorn [32]. In Longhorn Microsoft intends to change the way graphical components are being built by introducing Avalon [33]. Avalon uses a markup language called
Extensible Application Markup Language (XAML) to describe GUIs.

The difference between the currently used GUI handler in Windows, Windows Forms and Avalon is the markup language XAML, which allows the developer to mix the markup code with programming code.

In C#, a button is created by writing:

```csharp
Button btn = new Button();
Btn.BackgroundColor = Brushes.LightSeaGreen;
Btn.FontSize = new FontSize(24, FontSizeType.Point);
Btn.Content = "Calculate";
```

In XAML the equivalent code will be written as:

```xml
<Button Background="LightSeaGreen"
FontSize="24pt"
Content="Calculate">
</Button>
```

Avalon is anchored on the .NET Framework and uses Direct3D for drawing graphics on the screen. Its architecture is built on the MVC pattern and keeps the presentation and event handling separated from the data.

3. Method

This paper will analyze the MVC pattern and look at how it can be used to simplify the development of GUIs. The frameworks Morphic, Ion and AppKit will be studied to see how they build and differ from MVC. The implementation part, called MorphR, will be built in Ruby, using a modified version of MVC as design model and with OpenGL as backend for the graphics. The purpose for this is to make the implementation as platform independent and flexible as possible.

The implementation will be built incrementally by adding one widget at a time in order to be able to test the basic functionality first and then add more functionalities and widgets as the implementation proceeds.

4. Analysis of GUI design alternatives

In order to produce a clear view of how to design the MorphR application studies have been done on similar applications and their designs. Focus has been put on how to create a button shown in a window. First a UML diagram of the MVC pattern is presented and then UML diagrams for Morphic, AppKit and Ion. Based on those diagrams, the design of MorphR evolved. For simplicity the UML diagrams below are just showing closely related classes and not the whole class diagram.

4.1. Design of MVC

If a window and a button were created using pure MVC, there would have been a clear distinction between the model, the view and the controller as shown in the diagram. User interface events are passed through the Mouse or Keyboard classes to update the Model or the View or their subclasses.

![Figure 2. UML class diagram of a MVC design.](image)

In this model, the data is represented in the class `Data`, which inherits from `Model`. The visualization of the data is represented in the class `View`, which has the subclasses `Window` and `Button`. Finally there is a `Controller` class with the `Mouse` and `Keyboard` classes as subclasses. In this way it is easy to change the presentation of the data and also the controlling functions without having to alter the data itself.

4.2. Design of Morphic

Applications in Morphic can be built using the MVC pattern, but they do not have to. For example user interface events are predefined in the Morph class and a submorph can either use the general event method or define a method of its own. The main distinction between this approach and the MVC pattern above is that the event handling can be performed inside the morph and not through external classes.

The class diagram below is showing a simplified class structure needed to add a button to a window in Morphic.
As in most object-oriented languages, everything is an object and therefore all classes inherit from the `Object` class. The class hierarchy is going from general to specific tasks in order to reuse as much code as possible. For example a `BasicButton` is a bordered, rectangle morph, where only the methods specific for a button is written in the `BasicButton` class and other methods that are general to other classes as well is written in higher classes.

An application in Morphic can consist of a `MorphWorldView` with a `MorphWindow` as its submorph. The window is then adding its own submorphs i.e. buttons, which in turn can have their submorphs in form of images or labels.

### 4.3. Design of AppKit

AppKit is designed by NeXT [16] and used as foundation in GnuStep [17] and Ion [15] frameworks. It has an event handling that is similar to Morphic’s, where the events are inherited form an event class, but it also uses two objects to represent a component i.e. a button, both the `NSButtonCell` and the `NSButton` is needed to represent a button. The reason for this is to be able to use multiple `NSCell` objects in for example a table. This can be compared with the MVC pattern idea to use multiple views of the same model or presenting multiple models in one view.

The user interface events in AppKit are handled by an ordered list in the `NSResponder` class. Any object that needs to handle events has to inherit form `NSResponder`. The `NSApplication` object is handling a list of all windows in the application. Each `NSWindow` object maintains a hierarchy of `NSView` objects. The window is distributing events to its views and provides a drawing area.

In contrast to the Morphic concrete `Morph` class the `NSView` class is an abstract class and subclasses implements a drawing method using graphics methods.

### 4.4. Design of Ion

According to the author of Ion, Steve Dekorte, the design is influenced by AppKit, but there is also many similarities to the Morphic design. The `View` class is a concrete class that handles its own events. Windows can add for example buttons as subviews. Since Ion currently is much smaller than Morphic, it does not contain general classes like Morphic’s classes for
bordered objects and rectangles. The windows and buttons inherits directly from the View object generating duplicate lines of code, but also a more flexible implementation of the subviews when all of them is providing its own drawing instructions.

This makes it easy to produce vector-based themes to alter the graphical presentation without changing the functionality.

Like in the other designs, events are handled by the morphs and each morph is able to maintain a list of their submorphs.

5. Design of MorphR

Since the purpose of this development was to design a small, but flexible framework, design ideas from all of the above designs have been considered while deciding which design best suited MorphR. As the name is implying, Morphic was the first design to consider, but Ion’s pure design made that design easy to understand and modify.

In order to make MorphR as platform independent as possible and yet powerful, it is written in Ruby with OpenGL for the graphics handling. These choices also gave the application good flexibility and the ability to do very much with a minimum of code.

The design is also intended to be easy to expand and modify in the future. To accomplish this all classes have their own specific task and are easily modified or exchanged by new classes. The appearance of the components, like the buttons and text fields, are drawn in an external theme class with drawing instructions.

A theme can be set to provide customized appearance of an application. The theme classes is also hierarchical and the StandardTheme class is containing general definitions of the drawing methods while i.e. the WinterTheme class only defines those classes that differs from the ones in the
MorphR – A Morphic GUI in Ruby

StandardTheme. The ResourceManager object handles the themes and also the fonts.

The Mouse object stores information from the Screen class on what mouse button has been clicked and what target has been hit. Other classes can then ask the Mouse class for that information.

6. Results

The MorphR framework is using the basic ideas mostly from Morphic and Ion. The objects shown in the screen are called morphs, and can receive position, size and text as inputs when created or at runtime. All Morph classes have been built by vector graphics in OpenGL by specifying points and drawing lines between them.

In order to show an example of MorphR, a sample quiz program has been implemented with a window containing a label, a text field, a progress bar and buttons. When clicking the buttons, the corresponding letter is shown in the text field, or the text field is cleared, the answer checked, a new question shown or the application closes.

6.1. Design

With this design the application is created by creating a screen as parent or supermorph and add a window as a submorph. Then the other morphs like the label, the text field, the buttons and the progress bar is added as submorphs to that window, specifying the size and positioning of all morphs.

6.2. Morphs

Any component shown in the screen is a morph which inherits general defined methods from the Morph class. Any specific method that a morph requires is defined in that morphs class.

All morphs have their own definition of events and their drawing methods are either specified in the morph or is calling a theme drawing method. If a theme is not specified, the standard theme is called.

Currently widgets for button, text field, label and progress bar have been implemented as morphs. These widgets can be considered as the core components of many applications and other components are easily added since the basic functionality already is defined in the Morph class.

6.3. Timers

MorphR is designed with the possibility to use timer effects. This sample quiz program sets the timer when the user clicks the start button by calling a method in the Screen class that adds a timer to a specific target and that calls that targets timer method with a specified delay. OpenGL supports the use of multiple timers, but in this implementation, there is only possible to add one timer.

The timer effect is exemplified as a progress bar. The progress bar shows how much time there is left to answer the question. When the time is up, a new question is shown, the text field showing the answer is cleared and the progress bar is restarted. The progress bar is also restarted if the user wants to see a new question by clicking the start button.

6.4. Themes

Two themes have been implemented to show how the appearance easily can be altered. The white buttons are using a StandardTheme, while the letter buttons are using a Winter Theme. Currently the only method in the Winter Theme is the front color of the buttons. All other methods are inherited from the Standard Theme class.

The theme classes is called from the button draw method and the theme is specified when the button is created. It is also a possible to change the theme at runtime.
7. Discussion

This early implementation of the MorphR framework is only scratching the surface of what is possible to build with this kind of framework. It provides flexibility and powerfulness when it comes to build and customize a GUI.

In Morphic, which have been a large inspiration in this paper, there are some features that can be useful, but are not implemented at this point. The drag and drop functionality and the accessibility through halos, as shown in Figure 1, are not implemented at this moment, but can be very useful when altering a morph.

While working with the theme development, two different design methods were examined. One design used the factory design pattern and modules in Ruby, but this design was rather abstract and complex for a small sample program like this. Instead a design with theme access through a ThemeManager class was chosen. There is a possibility to use the module alternative in an extension of this framework.

During the work with this implementation the most difficult issues have been to transform the theories from models to reality. They are straightforward to understand, but not as easy to implement in practice.

The application grew incrementally during the process, which started with the core functionality of a button and a text field. Then mouse functionality and a label widget were added along with the font and timer classes. By working like this, the core functionality remains to provide the basics and additional classes and functions can be added one by one. This makes it easy to continue the development of this framework.

8. Future Work

During the initial development of the framework, mouse interaction was prioritized, but naturally the keyboard functionality is also prepared for. Currently the ESC button works for closing the application.

Timers are currently implemented in the way that it is possible to set one timer per application. OpenGL has support for adding more timer functions in each application, which gives possibilities only limited by the developer’s imagination.

Initially, widgets for buttons, text fields, labels and progress bars have been implemented as core widgets. Most applications use some or all of these widgets and each new-implemented widget highly increase the functionality.

In order to be able to change fonts and font sizes it is prepared to read in .bdf files, containing instructions on how to draw font characters, thus making it easy to add new fonts, both bitmapped and vector-based to this framework.

9. Conclusion

This paper aimed to implement a flexible, platform independent and graphical framework in Ruby. The design and the implemented classes are developed to simplify further development and future implementations of more widgets and functionality.

The visualization of timers is here used in terms of a progress bar that changes the question when the time is up. Timers are one thing that really makes Morphic stand out from other kinds of graphical frameworks and therefore timers are used in the MorphR framework too.

The appearance of an application is also important and themes are used in order to be able to customize the look and feel of MorphR.

We believe that this paper shows that frameworks like MorphR have large potential when it comes to produce flexible, functional, customizable and user-friendly GUI applications.

10. Acknowledgements

While working with this paper help, support and encouragement, both with the structure of this paper and also with the implementation of the framework, has been provided by Robert Feldt and Richard Torkar at the University of Trollhättan/Uddevalla, HTU. Your fast replies and qualitative help has been very appreciated. Samantha Jenkins at HTU has also provided thoughtful and valuable comments throughout the project.

A thank you should also be sent to Steve Dekorte, the author of Io for taking the time to quickly answer questions about Ion.

11. References


[16] NeXT’s AppKit documentation, *Cocoa: Application Kit (Objective-C)*,

[17] Introduction to GnuStep, *Introduction to the AppKit*,


