Developing a Graphical Library for a Clojure-based Introductory CS Course

Paul Schliep, Max Magnuson, and Elena Machkasova
Computer Science Discipline
University of Minnesota Morris
Morris, MN 56267
schli202@umn.edu, magnu401@umn.edu, elenam@umn.edu

Abstract

This project is a part of an ongoing effort on adopting the programming language, Clojure, for an introductory college-level CS course. Clojure is similar to the programming language Racket currently used in an introductory class at UMM, but provides better parallel processing and integration with other programming languages that would benefit students in their future careers. The objective of this project is to develop a beginner-friendly graphical library that focuses on functional approaches to developing programs. We have developed elements of a graphical library that provides the desired functionality, while maintaining a focus on engaging students and key concepts of an introductory CS course, such as problem solving and modularity. We hope that the developed graphical package, once completed, will be used for an introductory CS course.

1 Introduction

UMM offers an introductory CS course on teaching the concepts of problem solving and programming using the functional programming language Racket, which is a member of the Lisp family of programming languages. Racket is specifically designed for students new to programming [4]. Functional programming is a programming style of building elements of programs while retaining immutable data structures and without directly manipulating memory or changing state. Imperative, on the other hand, is also a programming style which uses a sequence of statements to build a program using memory manipulation and changing the state of objects in a program. The course utilizes functional approaches of Racket in order for students to better learn the general paradigms of programming. These functional approaches work well to teach students new to programming since functional approaches encourage programming without side effects and emphasize core CS concepts such as recursion. However, since Racket is not often used in real world settings, it does not benefit students as much in their future CS careers. Clojure, which is a functional language also in the Lisp category, offers better support for concurrency, integration with Java, and is gaining popularity in industry, thus it is a promising candidate for an introductory CS course. We plan to integrate Clojure into the introductory course in place of Racket because of the benefits it offers, but in order to do so, we need to resolve some of its limitations. One such limitation is Clojure's lack of a fully functional graphical library suitable for an introductory CS course.

Racket has a full graphical API that is useful for students to create a wide array of programs and also remains consistent with the functional design. Since graphics are a key motivating component in an introductory CS course, it is important that we implement a graphical API with user-friendly functions for introductory CS students. To fill the gap, we are integrating Quil, an open source graphical environment, into the course [1]. Quil shows promise since it has a large feature set appropriate for introductory students to create an extensive array of graphical programs, such as creating a drawing of concentric circles, an animation, or a game. However, Quil is built on top of the Java Swing library and has an imperative design approach to creating programs that take students away from functional approaches. It is important that functional approaches remain consistent throughout the course, and having a graphical library with an imperative feel would not be ideal for teaching beginning CS students. So, to keep the consistency of Clojure's functional approaches and continue teaching the key concepts of the introductory course, we are creating a graphical set of functions built on top of Quil's API that has similar functional approaches as Racket's graphical API. In order to do so, we abstract over many of the functions offered by Quil. While there have been many technical difficulties in working with Quil on creating this graphical set of functions, we have been able to successfully abstract over many of Quil's current functions to create a system that is closer to the functional design of Clojure.

2 Overview of Clojure

Clojure is a functional programming language in the Lisp family of languages. Clojure was developed by Rich Hickey and released in 2007 [5]. Clojure was developed with a strong emphasis on functional programming with its immutable data types and first class functions. Additionally, Clojure provides a rich set of immutable data structures, such as lists, vectors, and hashmaps.

Immutable data types are data types that cannot be changed. In Clojure when a change to an item of data is needed, a new data item will be made with that new value. Immutable data types are useful for avoiding side effects in functions. A side effect is when a function directly manipulates memory or noticeably interacts outside of its own scope other than to return a value. So if a function would change a global variable then it would have a side effect. Side effects can make finding the cause of a problem in code more troublesome. If a function interacts with more than itself then the problem can be spread out through the rest of the program making it more difficult to resolve. By reducing side effects, the problems will often be localized and easier to fix.

Clojure's syntax is similar to other Lisp languages. It uses prefix notation. Which means, all functions take the form of

```
(<name of function> <argument 1> <argument 2> ...)
```

This form even includes mathematical operations such as addition.

```
(+ 2 2)
-> 4
```

The symbol `->` denotes the result returned by the Clojure interpreter.

Clojure supports anonymous functions. This allows programmers to make functions on the fly when needed. For example, we may need to create a function to square a number.

```
(fn [x] (* x x))
```

Here the `fn` states that we have an anonymous function. Anything in the brackets following the `fn` are the arguments that the function takes, in this case `x`. After that is the body of the function which in this case is a simple multiplication of the argument `x` to itself. If we wanted to be able to refer to this function later on in our code then we can use `def` to give it a name.

```
(def square-root[x] (* x x))
```

Now this function can be reused anywhere in the code by calling `square-root`.

First class functions are functions that can be passed as arguments into other functions, returned from functions, or stored in data structures.

```
(map square-root [1 2 3 4])
-> [1 4 9 16]
```

Here `square-root` is passed in as an argument to `map` which takes a function and a collection as arguments and applies the function to each item in the collection. It then returns the resulting collection `[1 4 9 16]`.

Another important data structure in Clojure is hashmaps:

```
{:a 1 :b 2 :c 3}
```

A hashmap is collection of key-value pairs. The keys in the hashmap are directly connected to a value. Typically keywords are used as keys in hashmaps. Keywords are simply identifiers of the form `:a, :name, :student`. In the hashmap above the keyword :a is bound to the value 1, the keyword :b is bound to the value 2, etc.

3 Goals and Setup for an Introductory Course

3.1 Overview of Current UMM CS Introductory Course

CSci 1301 *Problem Solving and Algorithms Development* is an introductory course for CS majors and minors. Students are not expected to have any background in computing prior to the course. The course focuses on algorithmic approaches to problem solving and implementing solutions in a programming language. For over 15 years the course has been utilizing a dialect of Lisp: first Scheme, and then a very similar dialect Racket that has been specifically developed for teaching [4]. Racket comes with a beginner-friendly development environment, DrRacket, which introduces students to the language level-by-level, making available more and more language capabilities as students move through the course. While both Scheme and Racket provide optional mutable data, the focus of the course is on functional approaches that do not utilize mutability.

The use of a functional language exposes students to concepts such as recursion by working with lists and abstraction by working with higher order functions and generalizing solutions to handle a wider and wider array of problems. For instance, students write a function to add 1 to each element in a list of integers, and then a function to convert each element of a list of strings to lower-case. After writing several functions that follow the same pattern of applying an operation to each element in a list, they arrive at a concept of `map`: a higher-order function that takes a list and a function and applies the function to each element of that list. This type of generalization and abstraction is essential later in an algorithms course, in a course that introduces object-oriented design, and in many other areas of CS.

One of the important elements of the current CSci 1301 setup at UMM is open-ended assignments in which students design interactive graphical programs via Racket's library that allows manipulating and displaying the state of a game (the so-called "world") in a purely functional way. The set of predefined functions allows students to create simple animations as early as less than a month into a semester. Later in the semester students create a multi-object interactive game in which objects can be controlled by keyboard characters, such as arrow keys, or mouse clicks. The ease with which shapes and images can be incorporated into Racket adds to the entertainment factor of the game. The course emphasizes group work which prompts students to discuss approaches to their design and encourages them to improve their coding style.

3.2 Plans for Introducing Clojure

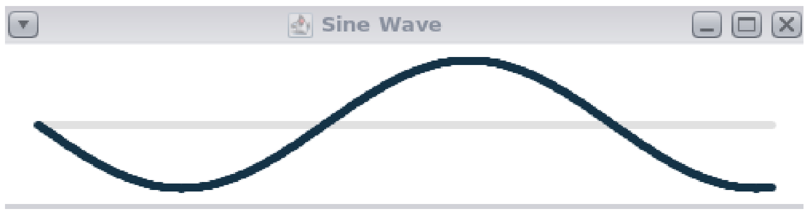
Racket's ease of use and functional design are helpful for teaching the learning objectives of an introductory course, but since it is not often used in the industry, it is not an ideal language for students to learn for their future CS careers. Clojure, however, has more compelling benefits for students in their future CS careers such as integration with Java, better parallel processing, and increasing popularity in the work field. What also makes Clojure well-suited is that it has functional design similar to that of Racket. Functional design in an introductory course is important since it teaches students many fundamentals of programming and problem solving such as recursion, modularity, and the importance of immutability. The benefit of teaching these concepts using functional approaches over imperative approaches is that functional programming is without side effects because of its immutability and stateless design. Because of this stateless design and lack of side effects, students would not need to worry about managing in-place object changes or managing memory, thus avoiding a common cause of confusion. Functional programming is well-suited for learning problem solving as well since it teaches students to apply different approaches to solving problems and can help students engage in other languages more easily.

Although Clojure provides many important benefits not seen in Racket, it is a language not yet ready for use in an introductory CS course. Many steps must be taken in order to accommodate for Clojure's steep learning curve that we hope will provide alternative approaches with easier transitions to programming.

One barrier preventing Clojure from being used in an introductory CS course is its error messages. Often, they tend to be unintuitive and unnecessarily long and can be hard to understand for experienced programmers, much less beginning CS students. Also, there is not an environment that supports easy navigation of the error messages and connecting them to the location of the offending line of code (such as highlighting that line of code). Since problem solving is a core learning objective of the introductory course, it is essential for error messages to be usable for beginning students to enable them to easily troubleshoot issues and focus on key learning concepts [3].

Along with the lack of an appropriate user interface for error messages, there is also an absence of a complete development environment for beginner programmers to easily program in. Currently, there is a Clojure IDE that is still in development called Light Table [2]. Since Light Table was designed for programming in Clojure and was created with usability in mind, it shows promise in usability for introductory computer science students. We are currently exploring ways of integrating our error message handling with Light Table before it can be useful for an introductory course.

A graphical environment in an introductory course is useful since it provides good motivation for students to explore the language and practice their programming skills. However, there is currently no graphical library for Clojure that teaches the learning objectives of functional programming in an introductory CS course. A potential graphical library to use for programming is an open-source project called Quil. Quil provides a lot of functionality that is needed for a graphical library necessary for students to use in an introductory CS course. Below is an example of a drawing done in Quil picturing a sine wave.



However, there are limitations and potential issues to using this graphical library that would steer away from the learning objectives and functional approaches that UMM's introductory CS curriculum has in place. Our project's goals are to resolve these limitations posed by Quil and have a fully functional graphical library. We hope this graphical library will encourage students to create interesting and useful programs and still develop their problem solving skills through functional approaches.

3.3 Requirements for a Graphical Library

Racket's graphical library has proven to be a powerful teaching tool in the current UMM CS introductory course by allowing students to make their own animations less than a month into the semester without any prior programming background. This maintains interest in programming while reinforcing key concepts. Those key concepts are reinforced in part by the functional design of Racket's graphical library. It uses a well known approach called model-view-controller (MVC).

MVC is a way of handling user interfaces. The model contains the state of the system which is the collection of data used for displaying graphics. The model is also responsible for updating this collection of data. The view is the visual representation of the data. The controller listens for changes to the model and sends these updates to the model. Then, the view is updated accordingly.

In the game of checkers the model would be the collection of the positions of the pieces, the color of the pieces, and the board itself. The view is what actually displays the board and each of the pieces, and the controller is what takes in the user input of what move the user wants to make. In the process of the user making a move, first the message would be interpreted by the controller. Then, the message would be sent to the model, so that the position of the piece can be updated. Finally, the view will update to reflect the change in position of the piece.

The modularization of work implemented by the MVC greatly reduces the dependency on the order of operations. If everything was updated piece by piece by taking in input, updating the data, then displaying it graphically, then the order of operations of the model, view, and control would all matter simultaneously. Since the process is separated into independent components, the order of operations only matters within each component. Model-view-controller is the system used by Racket's graphical library.

State in Quil is a collection of data related to each object in the system, not the graphical representation itself. If we needed to draw a wheel on a car that was moving across the screen, the tire and the rim would have properties such as position and size that are needed for drawing the wheel. The state in this case would be the collection of position and size.

Drawing in Quil mashes the process of updating state and drawing state into a single function. In the wheel on a car example, first the tire's position would be updated and drawn, and then the rim's position would be updated and drawn. Even in this small example, the process is very dependent on order of operations. Two operations that are not inherently dependent on each other, updating the tire and updating the rim, are now dependent on the order in which they happen. This situation would be avoided by separating updating state and displaying state similar to how it is handled in the MVC framework.

4 Developing a Clojure Graphical Library

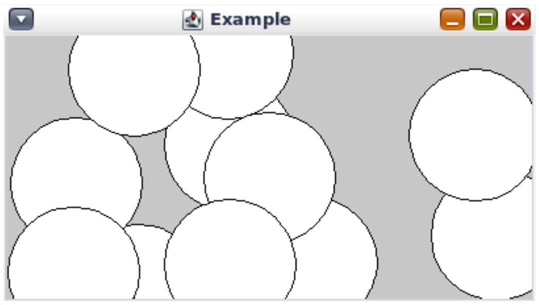
4.1 Overview of Quil

We are utilizing Quil to provide the needed tools for students to develop programs for graphical manipulation in Clojure. Quil is a graphical API developed for creating functions that display and transform graphical elements. It provides the user with a library of functions for creating shapes or changing colors that can be used for an array of projects for students in an introductory-level CS course. These projects can range from creating a drawing of repeating shapes to a game such as tic tac toe. Since Quil is built on top of Java Swing, it uses Java applets to display the graphical images and updates them constantly using a system of a set of frames that the user manipulates in order to animate drawings. However, Quil has some limitations that prevent it from being usable for an introductory CS course that teaches introductory students functional approaches to programming.

An apparent limitation of Quil is it has imperative approaches to creating programs that would not be acceptable for students to learn in a class based on functional programming. For example, when manipulating the world state in Quil, the state is first set, then the state is displayed and updated simultaneously in a draw function, which directly manipulates memory from the state and goes against the model-view-controller. In order to keep the consistency of the functional approaches and the MVC framework, we are working on creating a graphical library with design that implements functional approaches, abstracting over Quil's imperative design.

In our attempts to create the graphical library we have encounter many technical difficulties stemming from Quil's design. Developing programs in Quil's environment requires the user to make specific functions in order to correctly display the graphics. This involves creating a `setup` function for setting the initial background color, setting the frame rate, etc; developing a `draw` function for creating shapes and colors that will be transformed as necessary; and putting these functions into a `defsketch`, a macro that calls these functions as well as creating the title and size of the window. In order for Quil to be able to run programs, the user must use the `defsketch` for the work to be displayed. Below is an example of the typical setup in Quil consisting of the `setup`, `draw`, and `defsketch` functions. In this example, it creates an animation of repeated circles.

```
(defn setup []         ;; Setup function to be called in defsketch
  (frame-rate 1)       ;; Set framerate to 1 FPS, the speed it draws
  (background 200))    ;; Set the background color using RBG values

(defn draw []          ;; Draw function to be called in defsketch
   (ellipse            ;; Function that draws an ellipse
    (random (width))   ;; Set the x coordinate at a random width
    (random (height));; Set the y coordinate at a random height
    100 100))          ;; Set the diameters at 100.


(defsketch example     ;; Define a new sketch named example
 :title "Example"      ;; Set the title of the sketch
 :setup setup          ;; Specify the setup function
 :draw draw            ;; Specify the draw function
 :size [400 300])      ;; Specify the size of the window
```



Although Quil does feature some documentation on its functions, installation, and examples, it was still a challenge to learn how to use it because it was under-documented enough to where it caused some issues during our initial use such as understanding `defsketch` or the values for transparency in color. This also posed issues when making new functions that abstract over the current ones. This includes both the API's documentation as well as the documentation within the source code. So, when trying to understand how many of the functions worked (such as its system of state), we had to scour the source code and look at how the functions are set up to find a proper answer.

Our first step to create a more functional library was to separate the state manipulation so that it is called through three separate entities: set, update, and display. However, since Quil requires the user calling `defsketch` in order to call the functions of the program where the state is also manipulated, it posed issues on trying to create the desired functional system of state.

In order to overcome the challenges of developing our desired functional system state, we plan to hide the `defsketch` program by making it auto generated by a macro. This is necessary since `defsketch` takes the user out of the functional programming and would help us create the system of state where memory manipulation would be able to be more easily abstracted over.

5 Examples of Usage of the Graphical Library

In our design of the graphical library we needed to separate the processes of updating state and displaying state. In order to accomplish this we developed a system that would take in as setup:

- A collection of variables that describe state
- A collection of functions to update state
- An ordered collection of functions used to display graphics

These three collections correspond to each component of the MVC.

Example of user code before we abstracted over Quil's functions

```
(defn draw []
(draw-food)
(draw-snake)
(update))

(defn setup-scene []
(background-color "white")
(setup-state [:drawcoll [450 450 450 470 450 490 450 510]
:snakeHeadX 450 :snakeHeadY 450 :foodX 150 :foodY 150
:snake-direction "north" :foodExists false :score 0]))

(defsketch snake
:title "Hungry Hungry Snake"
:setup setup-scene
:draw draw
:size [900 900])
```

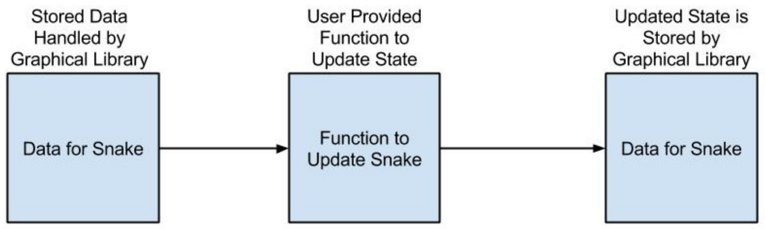
Example of user code in our graphical library design

```
(def states
{:snake [450 450 450 470 450 490 450 510] :snakeHeadX 450
:snakeHeadY 450 :foodX 150 :foodY 150 :snake-direction "north"
:foodExists false :score 0})

(def updates
{:setup-drawing setup :update-snake update-snake})

(def display-order
[redraw-canvas draw-food draw-snake])
```

This code is for a game of snake. The first example shows a fragment of user code from the original system before our version of the graphical library. In the first example, `draw` is a function that draws the food, then the starting position of the snake, and updates the snake when necessary based on user control of the keyboard. The `setup-scene` function is used to setup the initial state of the game and background-color. The functions within `setup-state` contain the data required for drawing the snake and for updating its position and size when necessary. Finally, the `defsketch` calls all of the required functions from the program and creates the window size and window title.

The second example shows the user code from our graphical library. `States` is a hashmap with data that is required for drawing the graphical representation of the game. `Updates` is a hashmap containing a setup function that will be run once followed by functions that are designed by the user that will be used to update the state. Finally, `display-order` is a collection of functions designed by the user to draw the different parts of the game in order. That order is the order in which the items are drawn.

Our system differs from Racket by how state is updated. In Racket the user takes in the entire state, and must break down the state into individual components before being able to update it or draw it, and then reassemble it. In our system, each portion of the state is tagged by keywords that match the function that update them. It is then supplied to the user-made functions at each stage of the MVC process. That means if a game contains snake and food, the user will provide a pair of functions for updating the snake and the food and another pair of functions for drawing them. The user maintains control over what constitutes components of state. We expect our system to be easier for students since they would not have to deal with the added complexity of having to deal with breaking down the state and reassembling it. Also a side benefit to our system is that functions won't need to be provide for parts of the state that do not need to be updated. This way students would not need to deal with updating parts of the state that do not need to be accessed.


Stored Data Handled by Graphical Library
User Provided Function to Update State
Updated State is Stored by Graphical Library
Data for Snake
Function to Update Snake
Data for Snake

This design accomplishes both abstracting over direct memory manipulation and making order of operations matter less. Our design abstracts over direct memory manipulation by taking care of changing variables for the user. When a part of the state is ready for updating, our system will input the stored data as an argument into the update function, then take the newly updated state and store that data. The accessing of the data and the storing of the data is all handled by the system and not by the user. When the graphics are displayed the data is accessed again by our system and given to the user-made function to draw the graphics.

6 Conclusions and Future Work

We have successfully abstracted over many of the functions and imperative approaches from Quil and have started to create a system that reflects the functional design of Clojure. As seen in the previous section, our example shows that our graphical library encourages a much more functional approach to creating programs and implements similar styles to Racket's graphical library in comparison to the original usage with Quil. With our system, students can intuitively create programs using functional approaches to produce graphical figures. This is made possible by using approaches similar to Racket's model-view-controller system. We've also abstracted over several of Quil's functions such as making shapes and inserting text that helps create a more intuitive system and a graphical library that encourages programming with functional methods.

While our developed graphical library accomplishes many of our design goals, it still has some work left before it can be considered ready for an introductory CS course. We plan to abstract over the `defsketch` function with our own macro in order to make it work with our developed code for state. This macro should be able to also create a system where students won't need to worry about calling their functions using a `defsketch` and can simply open a new project and begin making programs for displaying their work. We are writing documentation for our developed functions as well as examples to ensure students won't have difficulties understanding functions. We will continue to abstract over functions that we deem to be inaccessible to introductory CS students as we continue to develop the system.

References

[1] `https://github.com/quil/quil`.

[2]

[3] ELENA MACHKASOVA, STEPHEN J. ADAMS, J. E. Steps towards teaching the clojure programming language in an introductory cs class. presented at TFPIE, 2014.

[4] FELLEISEN, M., FINDLER, R. B., FLATT, M., AND KRISHNAMURTHI, S. *How to design programs: an introduction to programming and computing*. MIT Press, Cambridge, MA, USA, 2001.

[5] HICKEY, R. The clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages* (New York, NY, USA, 2008), DLS '08, ACM, pp. 1:1–1:1.